



JAVASCRIPT IMPRESSIONADOR

Curso Completo de JavaScript da Hashtag Programação



JAVASCRIPT IMPRESSIONADOR | HASHTAG PROGRAMAÇÃO

SUMÁRIO

MÓDULO 2

INTRODUÇÃO AO JAVASCRIPT

01	COMO VAI FUNCIONAR O CURSO	012
02	INTRODUÇÃO À PROGRAMAÇÃO	016
03	INTRODUÇÃO À LÓGICA DE PROGRAMAÇÃO - PARTE 1	022
04	INTRODUÇÃO À LÓGICA DE PROGRAMAÇÃO - PARTE 2	025
05	DESENVOLVIMENTO WEB E O QUE O JAVASCRIPT PODE FAZER	028
06	INSTALANDO O VS CODE	033
07	INSTALANDO O NODEJS	037
08	INTRODUÇÃO AO TERMINAL	038

MÓDULO 3

DADOS, VARIÁVEIS E OPERAÇÕES

01	INTRODUÇÃO AO MÓDULO	045
02	MEU PRIMEIRO PROGRAMA	046
03	REFINANDO MEU PRIMEIRO PROGRAMA	056
04	O QUE SÃO VARIÁVEIS	065
05	DECLARANDO E DEFININDO AS VARIÁVEIS – PARTE 1	068

MÓDULO 3

DADOS, VARIÁVEIS E OPERAÇÕES

06	DECLARANDO E DEFININDO AS VARIÁVEIS - PARTE 2	071
07	DIFERENÇAS ENTRE OS TIPOS DE DECLARAÇÃO	076
08	REGRAS E PADRONIZAÇÃO DE CÓDIGO	084
09	FORMATAÇÃO DE CÓDIGO	087
10	EXPLORANDO TIPOS PRIMITIVOS – FUNDAMENTOS E APLICAÇÕES	092
11	TIPOS DE DADOS – STRING NUMBER E NUMBER	093
12	TIPOS DE DADOS – BOOLEANS	105
13	TIPOS DE DADOS – UNDEFINED E NULL	109
14	OPERADORES ARITMÉTICOS - PARTE 1	114
15	OPERADORES ARITMÉTICOS - PARTE 2	118
16	OPERADORES DE COMPARAÇÃO	122
17	COERÇÃO IMPLÍCITA E MÉTODOS DE CONVERSÃO EXPLÍCITA - PARTE 1	127
18	COERÇÃO IMPLÍCITA E MÉTODOS DE COERÇÃO EXPLÍCITA - PARTE 2	131
19	OPERADORES LÓGICOS OU OPERAÇÕES BOOLEANAS – PARTE 1	134
20	OPERADORES LÓGICOS OU OPERAÇÕES BOOLEANAS – PARTE 2	137



SUMÁRIO

MÓDULO 3

DADOS, VARIÁVEIS E OPERAÇÕES

21	TIPOS REFERÊNCIAS	144
22	TIPOS DE DADOS – ARRAY – PARTE 1	148
23	TIPOS DE DADOS – ARRAY – PARTE 2	151
24	TIPOS DE DADOS – OBJETO – PARTE 1	157
25	TIPOS DE DADOS – OBJETO – PARTE 2	161
26	MANIPULAÇÃO DOS TIPOS REFERÊNCIAS	165

MÓDULO 4

FUNÇÕES

01	O QUE SÃO FUNÇÕES - PARTE 1	173
02	O QUE SÃO FUNÇÕES - PARTE 2	176
03	CHAMANDO FUNÇÕES	182
04	PARÂMETROS E ARGUMENTOS	185
05	VALORES DE RETORNO	189
06	MÚLTIPLOS PARÂMETROS	193
07	FUNÇÕES COMO EXPRESSÃO	197

MÓDULO 4

FUNÇÕES

08	FUNÇÕES DE ALTA ORDEM	201
09	FUNÇÕES PADRÃO E ARGUMENTOS OPCIONAIS	207
10	ESCOPOS DE VARIÁVEIS	213

MÓDULO 5

MÉTODOS

01	INTRDODUÇÃO AO MÓDULO	230
02	O QUE SÃO MÉTODOS?	232
03	DIFERENÇAS ENTRE MÉTODOS E FUNÇÕES	234
04	MÉTODOS NATIVOS – STRING – PARTE 1	237
05	MÉTODOS NATIVOS – STRING – PARTE 2	240
06	MÉTODOS NATIVOS – STRING – PARTE 3	243
07	MÉTODOS NATIVOS – ARRAY – PARTE 1	247
08	MÉTODOS NATIVOS – ARRAY – PARTE 2	251
09	MÉTODOS NATIVOS – ARRAY – PARTE 3	255
10	MÉTODOS NATIVOS – ARRAY – PARTE 4	258



SUMÁRIO

MÓDULO 5

MÉTODOS

11	O QUE SÃO PROPRIEDADES?	261
12	MÉTODOS NATIVOS – OBJETOS – PARTE 1	264
13	MÉTODOS NATIVOS – OBBEJTO – PARTE 2	268
14	MÉTODOS NATIVOS – OBJETO – PARTE 3	272
15	MÉTODOS PERSONALIZADOS	276
16	MÉTODOS NATIVOS – NUMBER – PARTE 1	281
17	MÉTODOS NATIVOS – NUMBER – PARTE 2	286
18	MÉTODOS NATIVOS – NUMBER – PARTE 3	290
19	OBJETOS GLOBAIS – CONCEITO	295
20	OBJETO GLOBAL – MATH – PARTE 1	297
21	OBJETO GLOBAL – MATH – PARTE 2	301
22	OBJETO GLOBAL – MATH – PARTE 3	304
23	OBJETO GLOBAL – DATE – PARTE 1	307
24	OBJETO GLOBAL – DATE – PARTE 2	310

MÓDULO 6

ESTRUTURAS CONDICIONAIS E DE REPETIÇÃO

01	INTRODUÇÃO / CONTROLE DE FLUXO DE EXECUÇÃO	315
02	BLOCOS CONDICIONAIS – IF	318
03	IF / ELSE IF / ELSE	323
04	SWITCH CASE	330
05	OPERADOR TERNÁRIO	337
06	ESTRUTURAS DE REPETIÇÃO	343
07	ESTRUTURA DE REPETIÇÃO – PARTE 2	344
08	ESTRUTURA FOR	346
09	FOR – EXEMPLO PRÁTICO	350
10	FOR – LOOP INFINITO	353
11	ESTRUTURA WHILE	356
12	WHILE – PRIMEIRA ABORDAGEM	359
13	WHILE – EXEMPLO PRÁTICO	361
14	ESTRUTURA DO... WHILE	365

SUMÁRIO

MÓDULO 6

ESTRUTURAS CONDICIONAIS E DE REPETIÇÃO

15	DIFERENÇAS ENTRE WHILE E DO WHILE	368
16	DO WHILE – PRIMEIRA ABORDAGEM	369
17	DO WHILE – EXEMPLO PRÁTICO	371
18	RECURSIVIDADE	375
19	O PROBLEMA DA RECURSIVIDADE	381
20	RECURSIVIDADE – EXEMPLO PRÁTICO (STRING)	383
21	RECURSIVIDADE – EXEMPLO PRÁTICO (OBJETO)	396
22	ENUMERANDO INFORMAÇÕES EM OBJETOS	390
23	ENUMERANDO INFORMAÇÕES EM ARRAYS	393
24	VALORES TRUTHY E FALSEY	396

MÓDULO 7

CONSOLIDANDO OS FUNDAMENTOS DO JAVASCRIPT

01	APRESENTAÇÃO DO PROJETO: LABORATÓRIO DE FUNDAMENTOS	399
02	RESOLUÇÃO DOS REQUISITOS – PARTE 1	400
03	RESOLUÇÃO DOS REQUISITOS – PARTE 2	405
04	RESOLUÇÃO DOS REQUISITOS – PARTE 3	408

MÓDULO 7

CONSOLIDANDO OS FUNDAMENTOS DO JAVASCRIPT

05	RESOLUÇÃO DOS REQUISITOS – PARTE 4	410
06	RESOLUÇÃO DOS REQUISITOS – PARTE 5	413
07	RESOLUÇÃO DOS REQUISITOS – PARTE 6	416
08	APRESENTAÇÃO DO PROJETO: SIMULAÇÃO FINANCEIRA	420
09	RESOLUÇÃO DOS REQUISITOS – PARTE 1	421
10	RESOLUÇÃO DOS REQUISITOS – PARTE 2	425
11	RESOLUÇÃO DOS REQUISITOS – PARTE 3	429
12	RESOLUÇÃO DOS REQUISITOS – PARTE 4	431

MÓDULO 8

CONSOLIDANDO OS FUNDAMENTOS DO JAVASCRIPT

01	COMO A COMUNICAÇÃO DA INTERNET FUNCIONA	436
02	ARQUITETURA E PROTOCOLOS DA INTERNET	441
03	O HTML	448
04	TAGS E SUAS INFORMAÇÕES	457
05	O CSS	461



SUMÁRIO

MÓDULO 8

CONSOLIDANDO OS FUNDAMENTOS DO JAVASCRIPT

06	PROPRIEDADES EM CSS	465
07	RPROBLEMAS DESSA PRIMEIRA ABORDAGEM	468
08	SEPARAÇÃO DE RESPONSABILIDADE: CRIANDO UM ARQUIVO CSS	471
09	1NLINE STYLING, SELETORES, CLASSES E IDENTIFICADORES	473
10	SOBRE TAGS HTML	476
11	ELEMENTOS DE NÍVEL DE BLOCO VS ELEMENTOS DE LINHA	482
12	SELETORES CSS	485
13	DO "BOX MODEL"	490
14	PRINCIPAIS PARÂMTEROS ESTILIZÁVEIS	496
15	APRESENTAÇÃO DO PROJETO LANDING PAGE - PIZZAHASH	502
16	RECURSOS EXTERNOS UTILIZADOS	503
17	ESTRUTURANDO O CABEÇALHO E O HOME	508
18	CRIANDO A SEÇÃO SOBRE	511
19	DESENVOLVENDO A SEÇÃO DO MENU	514
20	IMPLEMENTANDO A SEÇÃO DE PROMOÇÕES	517

MÓDULO 8

DESENVOLVIMENTO WEB

21	CONSTRUINDO A SEÇÃO DE AVALIAÇÕES	520
22	CONFIGURANDO A SEÇÃO RODPÉ	523
23	criando estilos globais	528
24	PERSONALIZANDO O ESTILO DO CABEÇALHO - PARTE 1	533
25	PERSONALIZANDO O ESTILO DO CABEÇALHO - PARTE 2	535
26	criando estilização do homt	537
27	APLICANDO ESTILOS NA SEÇÃO SOBRE	540
28	FORMATANDO A APARÊNCIA DA SEÇÃO DO MENU	542
29	ESTILOS PARA A SEÇÃO DE PROMOÇÕES - PARTE 1	545
30	ESTILOS PARA A SEÇÃO DE PROMOÇÕES - PARTE 2	548
31	ESTILIZAÇÃO DA SEÇÃO DE AVALIAÇÕES	550
32	CONFIGURANDO O ESTILO DO RODAPÉ - PARTE 1	553
33	CONFIGURANDO O ESTILO DO RODAPÉ - PARTE 2	556
34	RESPONSIVIDADE DA LANDING PAGE	559



SUMÁRIO

MÓDULO 9

DOM, FORMULÁRIOS E LOCALSTORAGE

01	A IMPORTÂNCIA DO FRONT-END · · · · ·	564
02	FRONT-END X CONSTRUINDO UM QUEBRA CABEÇA · · · · ·	566
03	DIFERENÇAS ENTRE FRONT-END E WEB DESIGN · · · · ·	572
04	ENTENDENDO O MÓDULO · · · · ·	574
05	O QUE É DOM? · · · · ·	575
06	O PROCESSO DE RENDERIZAÇÃO: UM PANORAMA GERAL [OPCIONAL]	579
07	EVENTOS HTML · · · · ·	582
08	SELECIONANDO ELEMENTOS NO DOM – PARTE 1 · · · · ·	589
09	SELECIONANDO ELEMENTOS NO DOM – PARTE 2 · · · · ·	591
10	SELECIONANDO ELEMENTOS NO DOM – PARTE 3 · · · · ·	595
11	SELECIONANDO ELEMENTOS NO DOM – PARTE 4 · · · · ·	603
12	HTML COLLECTION VERSUS NODE LIST · · · · ·	607
13	criando elementos HTML com JAVASCRIPT · · · · ·	614

MÓDULO 9

DOM, FORMULÁRIOS E LOCALSTORAGE

14	MANIPULANDO ELEMENTOS HTML COM JAVASCRIPT · · · · ·	620
15	DIFERENÇA INNERHTML E TEXTCONTENT · · · · ·	625
16	MODIFICANDO ATRIBUTOS COM JAVASCRIPT · · · · ·	629
17	RELACIONAMENTOS HTML: PAI E FILHO – PARTE 1 · · · · ·	634
18	RELACIONAMENTOS HTML – PAI E FILHO – PARTE 2 · · · · ·	642
19	INSERINDO E REMOVENDO ELEMENTOS · · · · ·	645
20	MANIPULANDO CLASSES CSS – PARTE 1 · · · · ·	650
21	MANIPULANDO CLASSES CSS – PARTE 2 · · · · ·	654
22	EVENTOS NO DOM : ADDEVENTLISTENER E REMOVEEVENTLISTENER	656
23	ATRIBUTOS VS. ADDEVENTLISTENER()	661
24	PREVENT DEFAULT COM ADDEVENTLISTENER: CONTROLE DE · · · · ·	664
	EVENTOS SIMPLIFICADO	
25	TRABALHANDO COM FORMULÁRIOS · · · · ·	666
26	EXPLORANDO OS ELEMENTOS DE FORMULÁRIO HTML – PARTE 1 · · · · ·	668

SUMÁRIO

MÓDULO 9

DOM, FORMULÁRIOS E LOCALSTORAGE

27	EXPLORANDO OS ELEMENTOS DE FORMULÁRIO HTML – PARTE 2	674
28	FORMULÁRIO COM EVENTOS EM JAVASCRIPT	678
29	MANIPULAÇÃO DE FORMULÁRIO	685
30	VALIDAÇÃO DE FORMULÁRIO COM JAVASCRIPT – PARTE 1	687
31	VALIDAÇÃO DE FORMULÁRIO COM JAVASCRIPT – PARTE 2	691
32	VALIDAÇÃO DE FORMULÁRIO COM JAVASCRIPT – PARTE 3	695
33	VALIDAÇÃO DE FORMULÁRIO COM JAVASCRIPT – PARTE 4	699
34	VALIDAÇÃO DE FORMULÁRIO COM JAVASCRIPT – PARTE 5	700
35	VALIDAÇÃO DE FORMULÁRIO COM JAVASCRIPT – PARTE 6	703
36	VALIDAÇÃO DE FORMULÁRIO COM JAVASCRIPT – PARTE 7	705
37	ENVIO DE DADOS - FORMULÁRIO	707
38	OBJETO WINDOW	712
39	INTRODUÇÃO AO ARMAZENAMENTO DE DADOS	718
40	LOCALSTORAGE – CONCEITO E ESTRUTURA	721
41	LOCALSTORAGE - SETITEM	724
42	LOCALSTORAGE – REMOVEITEM E GETITEM	728

MÓDULO 10

PROJETO HASHFORMS

01	APRESENTAÇÃO DO PROJETO	732
02	ESTRUTURA INICIAL DO FORMULÁRIO - HTML	733
03	FINALIZAÇÃO DO FORMULÁRIO- HTML	736
04	RESUMO E DESCONTO – HTML	738
05	GARANTIA E RODAP-E – HTML	741
06	ESTILIZANDO PÁGINA - ESTRUTURA GERAL	743
07	ESTILIZANDO PÁGINA - PARTE 1	747
08	ESTILIZANDO PÁGINA - PARTE 2	753
09	ESTILIZANDO PÁGINA - PARTE 3	756
10	ESTILIZANDO PÁGINA - RODAPÉ	760
11	VALIDAÇÃO DE DADOS BÁSICA - PARTE 1	762
12	VALIDAÇÃO DE DADOS BÁSICA - PARTE 2	765
13	SALVANDO DADOS DO FORMULÁRIO NO LOCALSTORAGE	773
14	TOGGLE DE FORMULÁRIO: CONTROLANDO EXIBIÇÃO DE CUPONS	776



SUMÁRIO

MÓDULO 10

PROJETO HASHFORMS

15	VALIDAÇÃO E APLICAÇÃO DE CUPONS DE DESCONTO	779
16	SALVANDO DESCONTO NO LOCALSTORAGE	784

MÓDULO 11

INTRODUÇÃO AO BACKEND

01	INTRODUÇÃO AO BACKEND	788
02	FUNCIONALIDADES DO BACKEND	789
03	FUNCIONALIDADE: SERVIDOR	792
04	LINHAS DE COMANDO (CLI)	794
05	CLI : INTRODUÇÃO AO NPM	798
06	CONHECENDO OS COMANDOS	801
07	INTRODUÇÃO: API E WEB API	806
08	OBJETO JSON: CONCEITO	808
09	PROTOCOLOS HTTP – STATUS CODE	812
10	EXEMPLO DE COMUNICAÇÃO FRONT-END E API – PARTE 1	813
11	EXEMPLO DE COMUNICAÇÃO FRONT-END E API – PARTE 2	820

MÓDULO 12

DEPURAÇÃO E DESENVOLVIMENTO EFICIENTE

01	OS VÁRIOS TIPOS DE ERROS	827
02	CONSOLE LOG: UMA PRIMEIRA ABORDAGEM	835
03	DENTRO DO NAVEGADOR	845
04	DEBUGANDO SEU CÓDIGO	864
05	DEV TOOLS NO VS CODE	874

MÓDULO 13

TRATAMENTO DE ERRO

01	INTRODUÇÃO Á PROGRAMAÇÃO DEFENSIVA	890
02	ANTECIPANDO ERROS COM CONDICIONAIS	893
03	TRY, CATCH E THROW	898
04	LANÇAR ERRO COMPLETO OU SÓ MENSAGEM?	903
05	FINALLY	905

SUMÁRIO

MÓDULO 14

PROJETO GERENCIADOR DE PLAYLIST DE FILMES

01	APRESENTAÇÃO DO PROJETO	908
02	RECURSOS EXTERNOS UTILIZADOS	911
03	COMEÇANDO A ESTRUTURA DO PROJETO	919
04	CRIANDO O NOSSO HEADER	926
05	ESTILIZANDO O HEADER	933
06	OS ELEMENTOS DE BUSCA	941
07	AJUSTANDO O HEADER PARA DISPLAY MAiores	946
08	SEÇÃO PRINCIPAL DA PÁGINA	947
09	CRIANDO A ESTRUTURA DO MODAL	952
10	ESTILIZANDO O MODAL	954
11	MOCK DO PRIMEIRO MODAL DE FILME	961
12	CSS DO MODAL PROJETADO	967
13	ADICIONANDO INTELIGÊNCIA AO MODAL	971

MÓDULO 14

PROJETO: GERENCIADOR DE PLAYLIST DE FILMES

14	COMEÇANDO A INTELIGÊNCIA DA PÁGINA PRINCIPAL	977
15	CAPTURANDO AS ENTRADAS DOS USUÁRIOS	979
16	CONSUMINDO A API DO OMDB	983
17	LIDANDO COM ERROS	988
18	NOTIFICANDO O USUÁRIO DE ERROS	991
19	MODAL DINÂMICO	995
20	PREPARANDO A LISTA DE FILMES	998
21	A INTELIGÊNCIA DO BOTÃO DO MODAL	1000
22	FINALIZANDO A FUNCIONALIDADE DE ADIÇÃO	1005
23	IDENTIFICANDO OS FILMES DE FORMA ÚNICA	1006
24	REMOÇÃO DE UM FILME	1008
25	AJUSTES FINAIS/ARMAZENANDO A LISTA NO NAVEGADOR	1011



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.

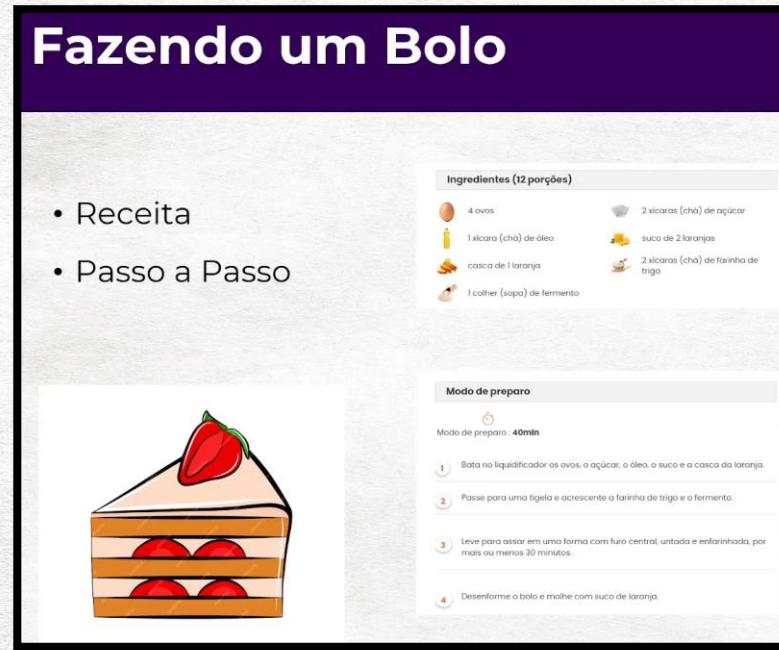
- **React Native:**

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

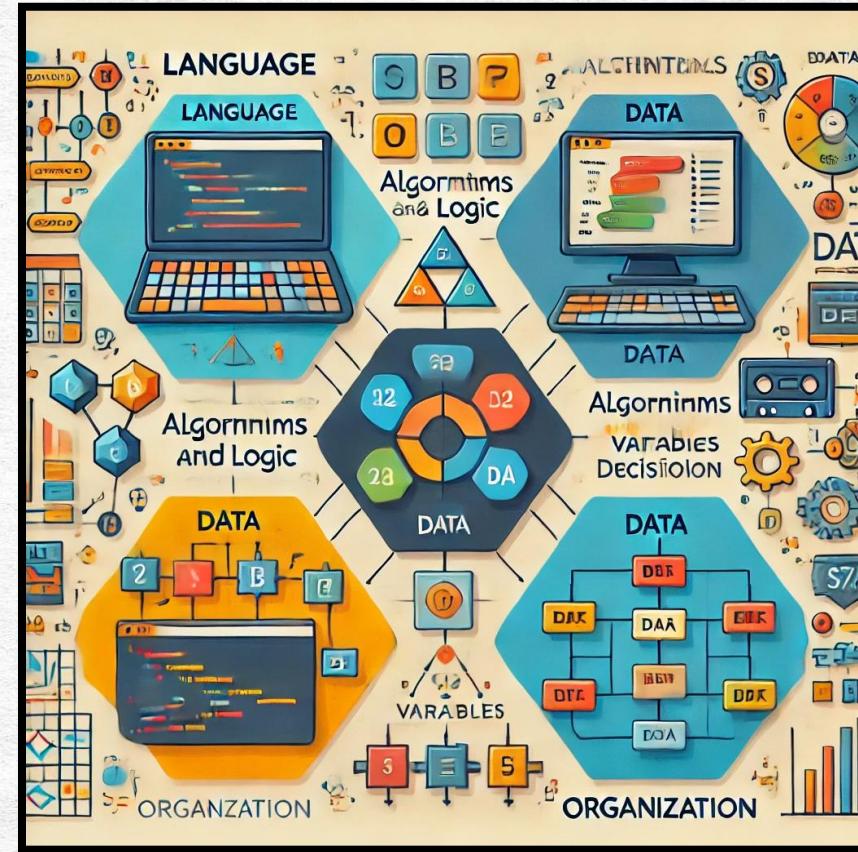
Os **dados** são o que alimentam os programas e representam informações do mundo real. Trabalhar com dados envolve:

- **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

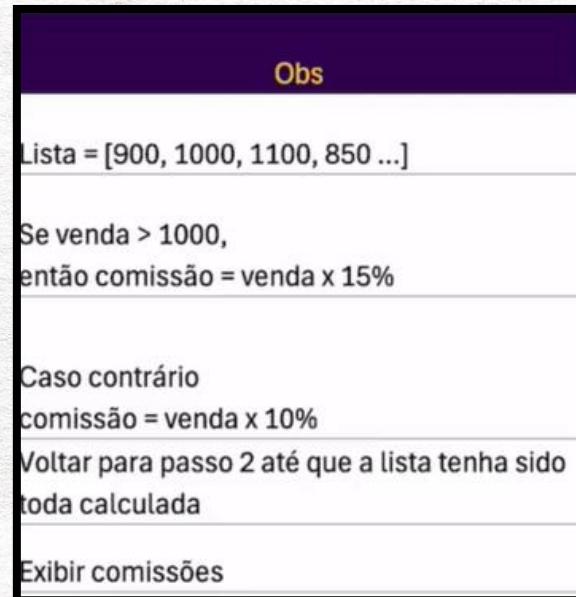


Planejamento do Algoritmo

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.

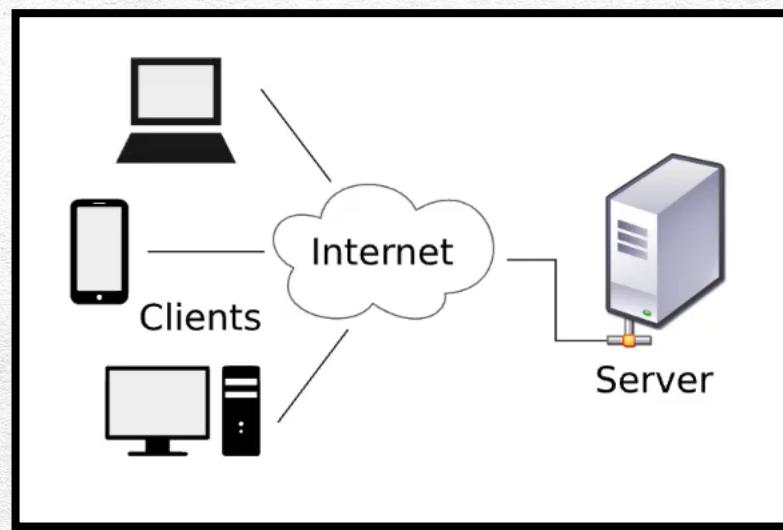
A comunicação entre o cliente (o navegador do usuário) e o servidor (onde a aplicação web está hospedada) é fundamental para o funcionamento de uma aplicação web. Esse processo geralmente ocorre através de um modelo de requisição e resposta, onde o cliente envia uma solicitação para o servidor e o servidor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



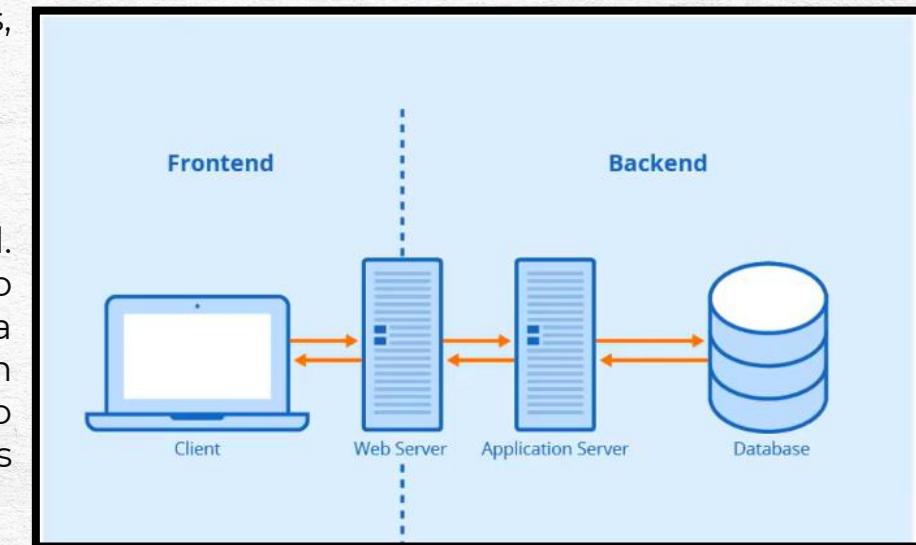
Aplicações Front-end

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

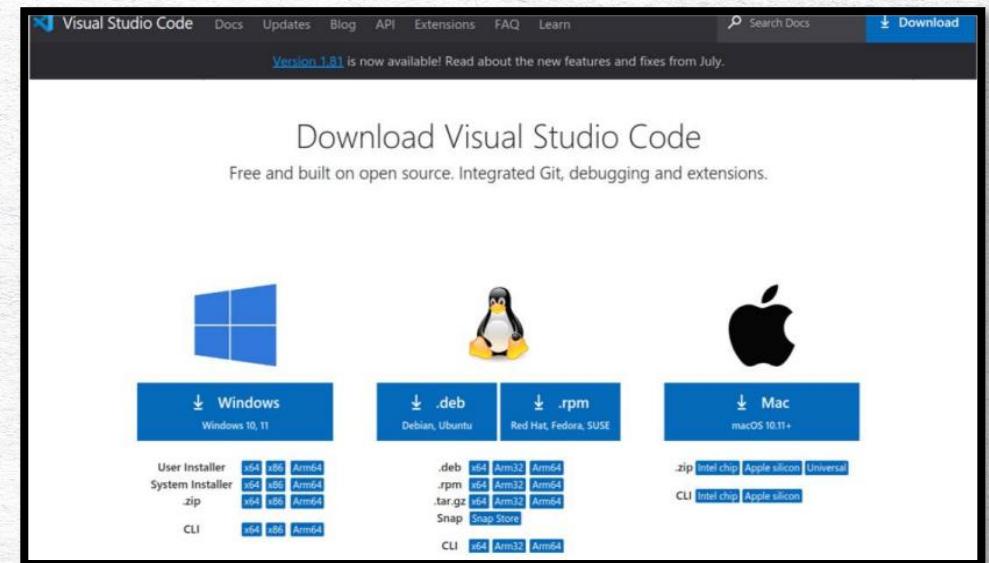
O Visual Studio Code, ou VS Code, é um editor de código que os programadores usam para escrever e editar seus programas. Ele é muito popular porque é fácil de usar e tem muitos recursos úteis. Um dos motivos para sua popularidade é o fato de que é muito bom para escrever JavaScript.

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/>

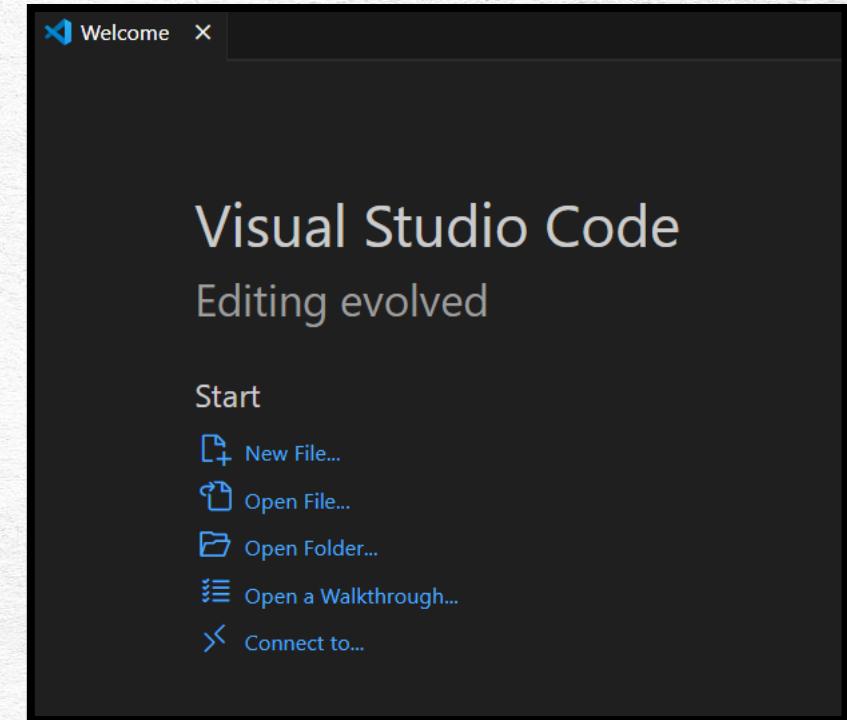
- 1 - Baixe o arquivo de instalação correspondente ao seu sistema operacional (Windows, MacOS ou Linux).
- 2 - Execute o arquivo de instalação e siga as instruções na tela. Em geral, é só continuar clicando em "Próximo".
- 3 - No Windows, durante a instalação, marque a opção "Add to path" para adicionar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



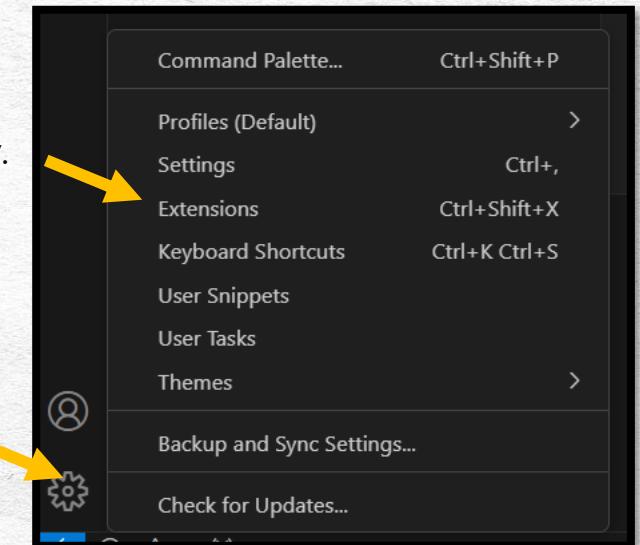
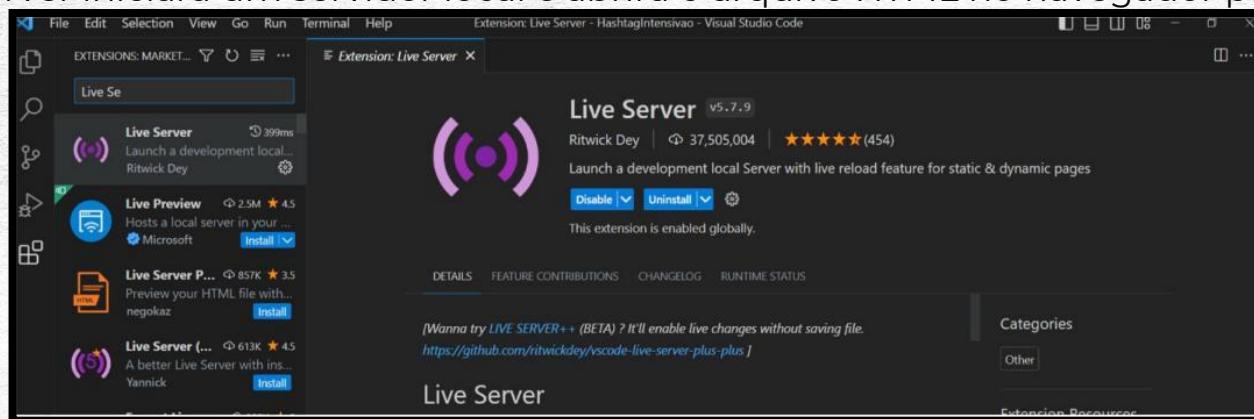
Extensões

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:

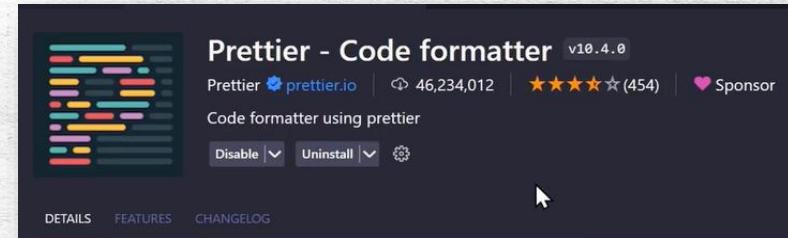
- Abra o Visual Studio Code (VS Code) e vá para a seção de extensões.
- Clique no ícone de extensões no menu lateral esquerdo (ou use o atalho "Ctrl+Shift+X").
- Pesquise por "Live Server" e clique em "Instalar" ao lado da extensão desenvolvida por Ritwick Dey.
- Aguarde a instalação e clique em "Gerenciar" 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.

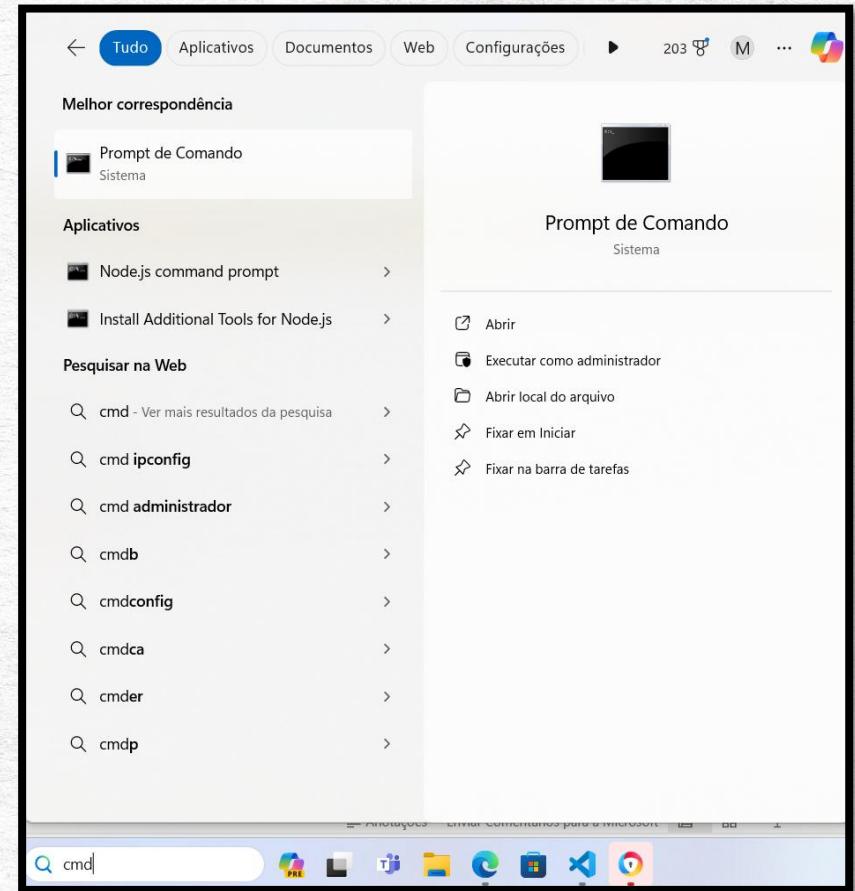
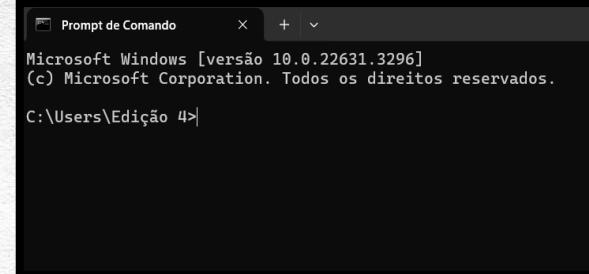
Aqui está uma breve explicação sobre cada um:

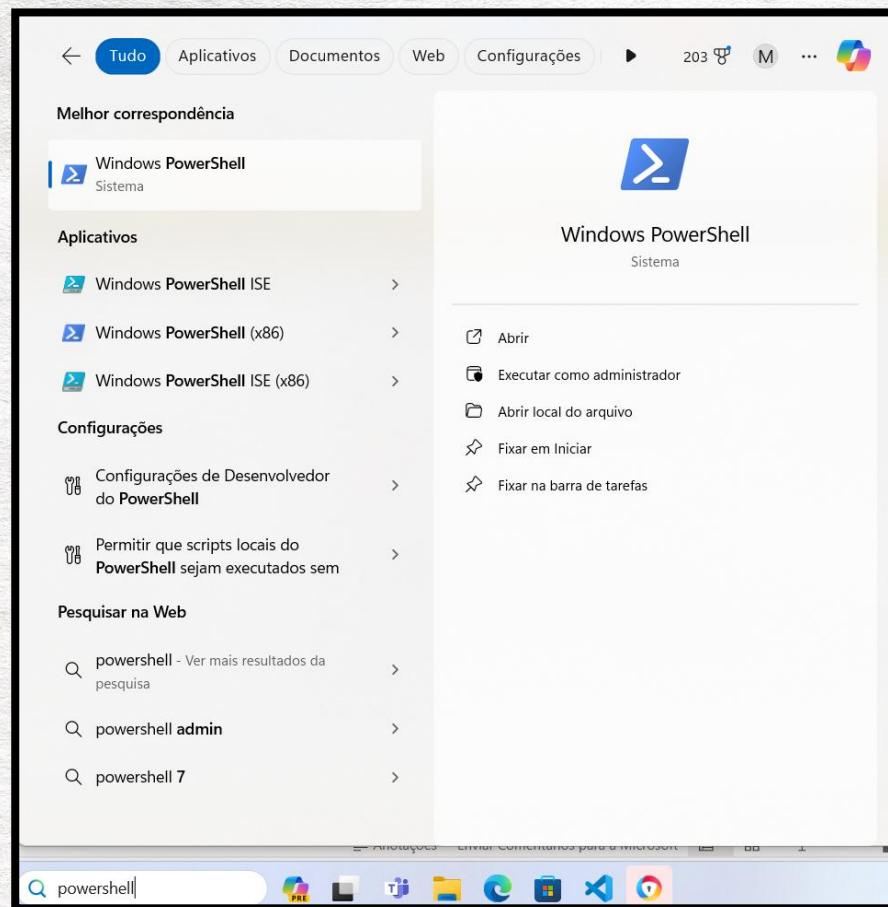
Prompt de Comando (cmd):

- 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
```

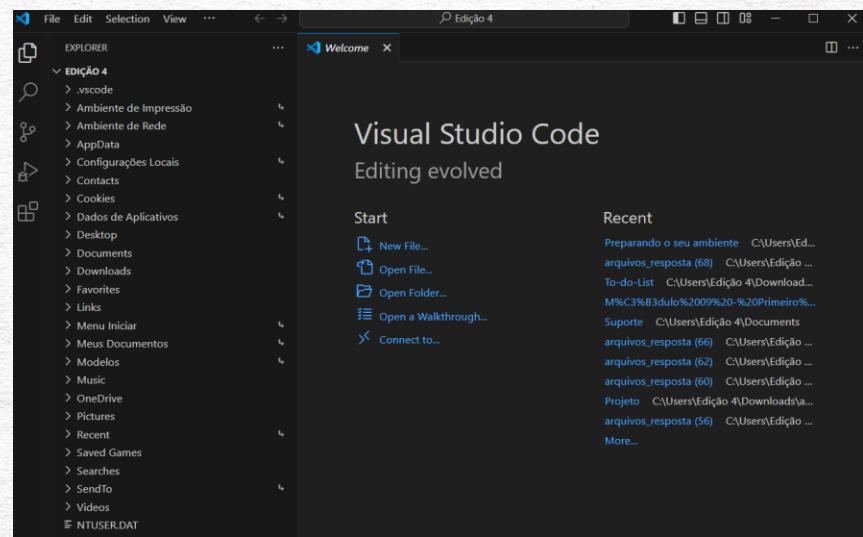
O comando "mkdir" (make directory) é utilizado no terminal para criar um novo diretório (pasta) no sistema de arquivos. Basta digitar "mkdir" seguido pelo nome do diretório que você deseja criar. Por exemplo, para criar um diretório chamado "novo_diretorio", você digitaria:

```
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:

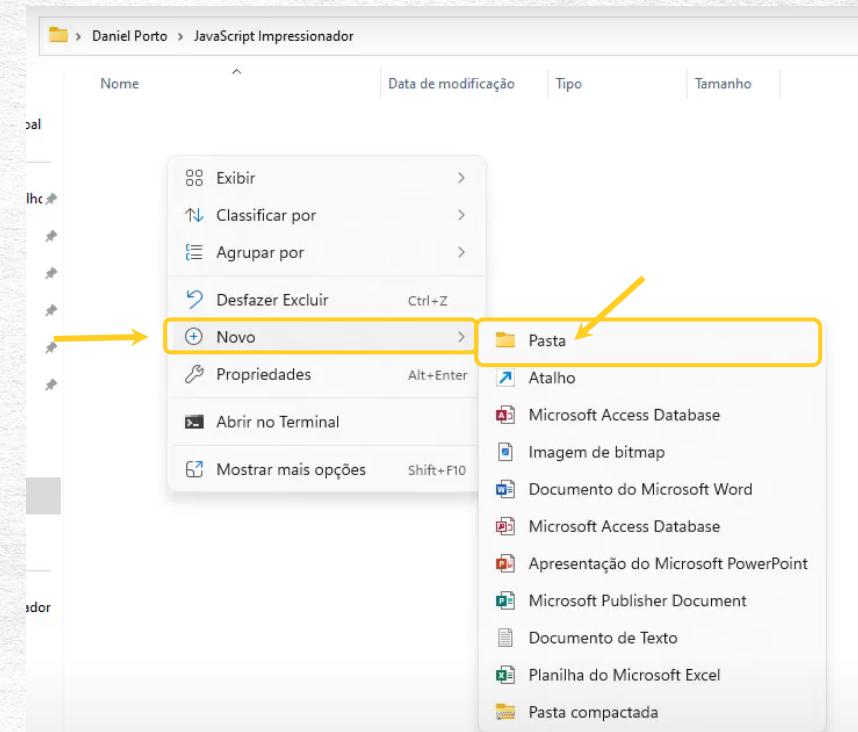
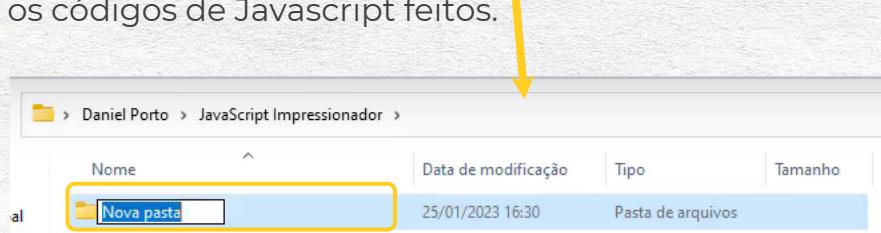
- **Números:** Representam valores numéricos.
 - Exemplo: 10, 3.14.
- **Strings:** Sequências de caracteres usadas para representar texto.
 - Exemplo: 'Olá, mundo! '.
- **Booleanos:** Representam valores verdadeiros ou falsos.
 - Exemplo: true, false.
- **Arrays e Objetos:** Estruturas que organizam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

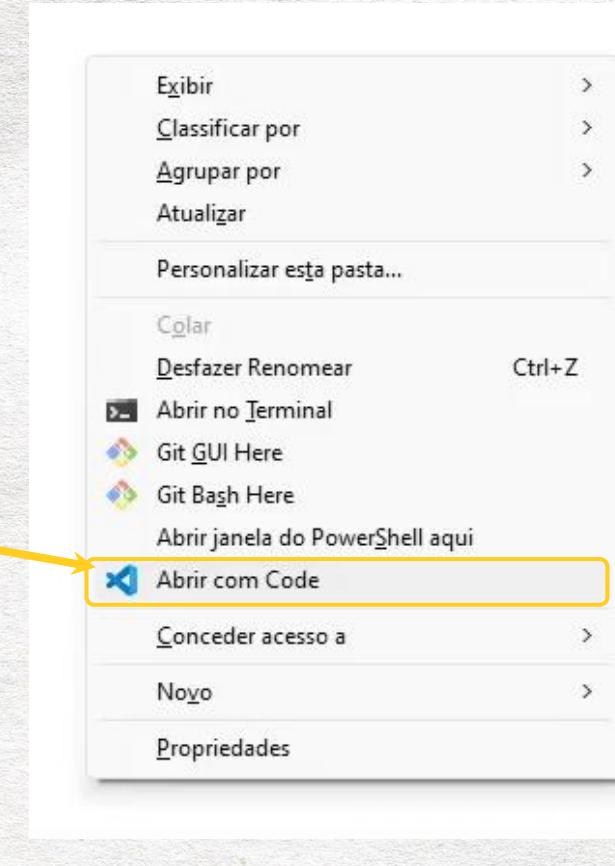
Esses dados são fundamentais para construir programas que interajam com usuários e realizem tarefas úteis.

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.

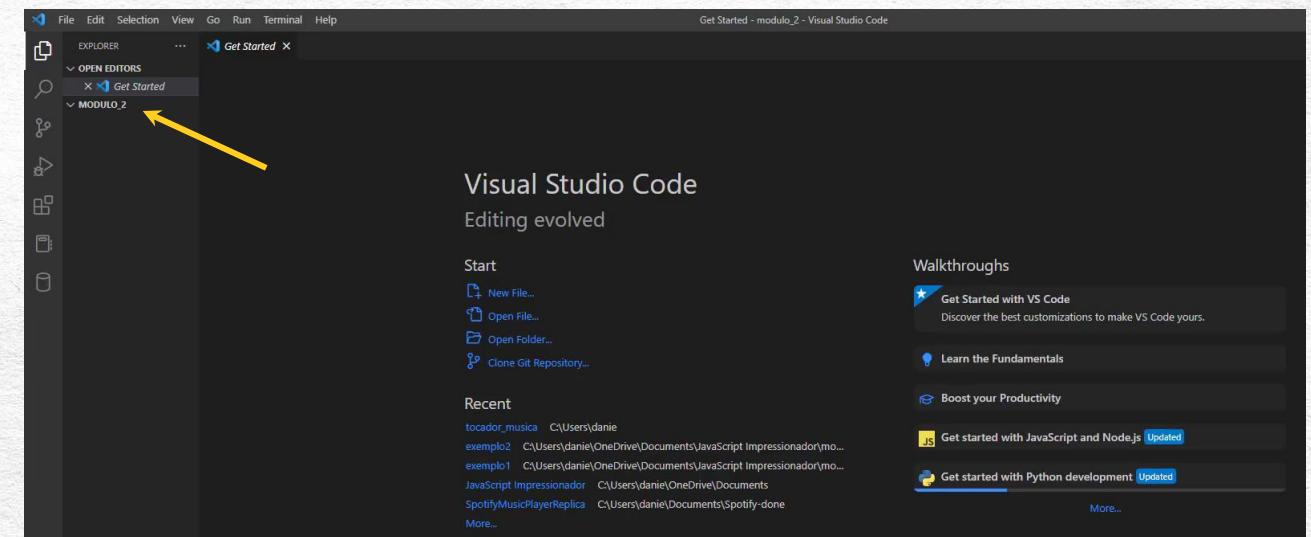
- **1º Passo:** Criaremos uma pasta chamada JavaScript Impressor no seu computador (no local que faça sentido para você- exemplo: Pasta Pessoal, Área de Trabalho, Documentos).
- **2º Passo:** Dentro da pasta JavaScript Impressor, clicando com o botão direito iremos clicar em ‘Novo’, depois ‘Pasta’, iremos 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.



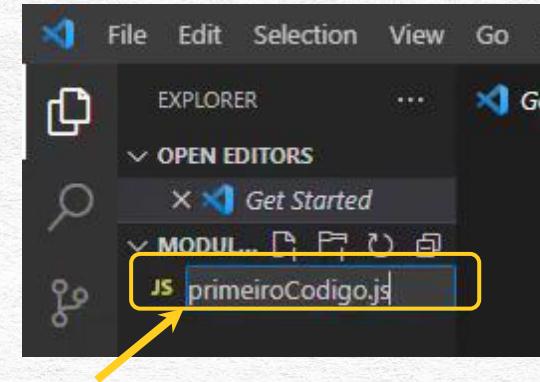
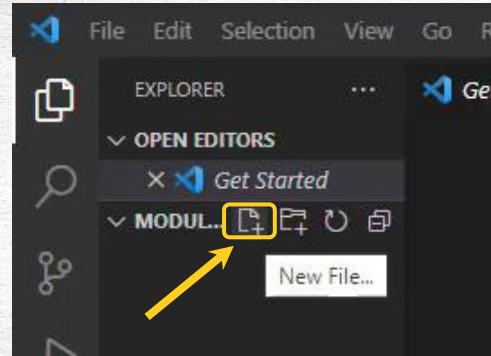


- **3º Passo:** Dentro da pasta modulo_2, iremos clicar com 'SHIFT + botão direito do mouse' e clicar na opção '**Abrir com Code**'.

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.



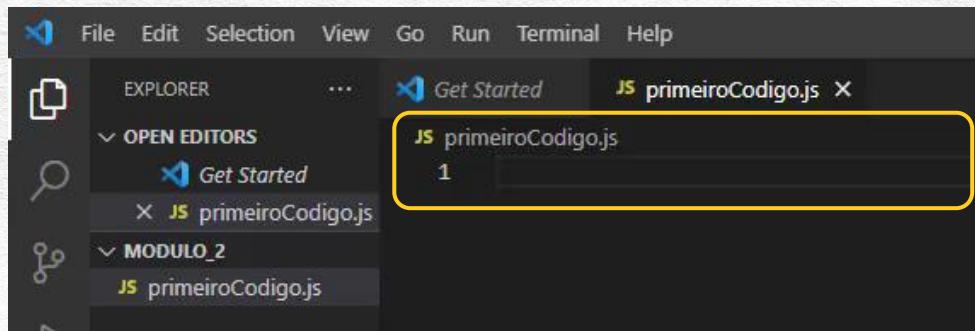
- 4º Passo: Iremos iniciar um novo arquivo, clicando no ícone de ‘New File’, nomeando-o “primeiroCodigo.js”



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??

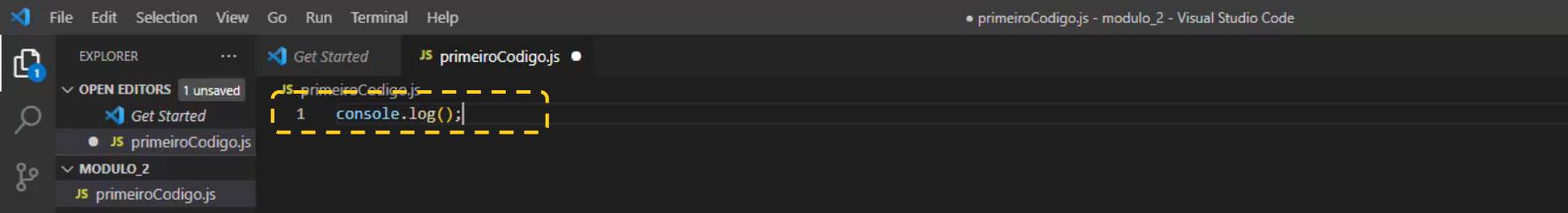
O VS Code, ao receber um arquivo “.js” automaticamente já faz a leitura de que se trata de um arquivo de código de Javascript, e isso ocorre porque essa extensão “.js” é utilizada para além de sinalizar, criar um arquivo do tipo de Javascript.

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.



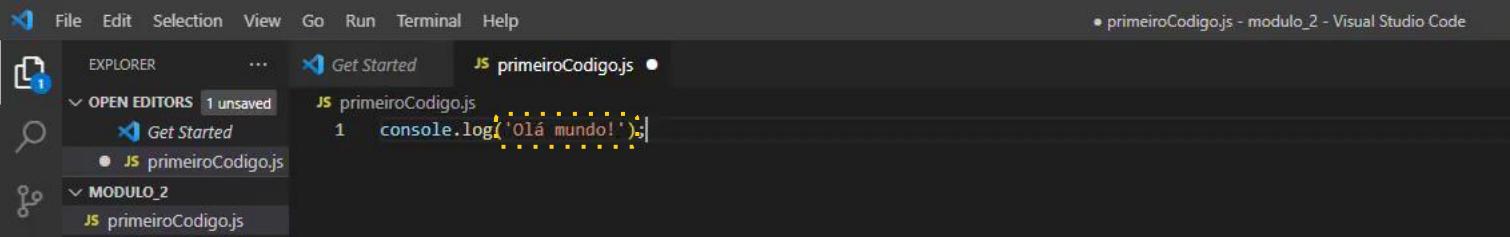
```
File Edit Selection View Go Run Terminal Help
EXPLORER ... Get Started JS primeiroCodigo.js
OPEN EDITORS 1 unsaved JS primeiroCodigo.js
Get Started
primeiroCodigo.js
MODULO_2 JS primeiroCodigo.js
```

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
```

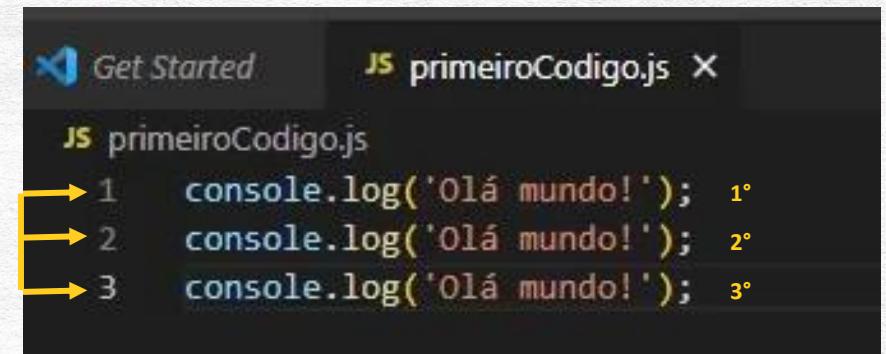
primeiroCodigo.js - modulo_2 - Visual Studio Code

```
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.

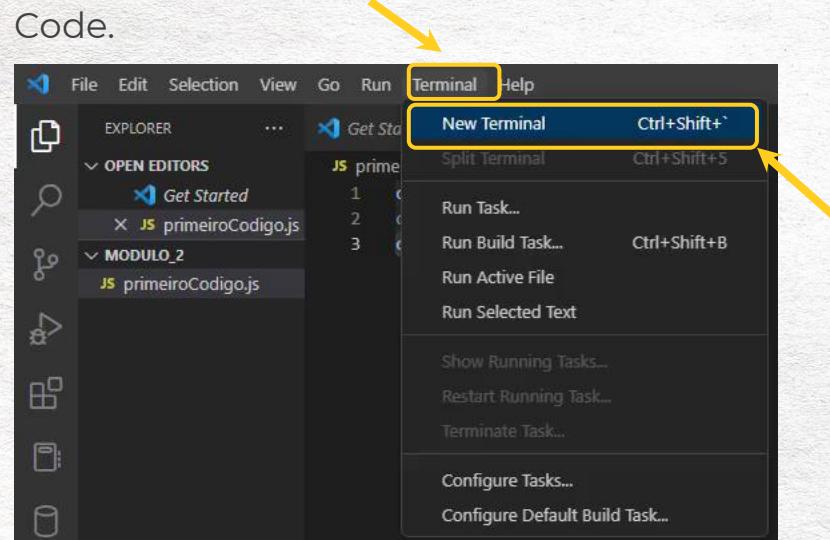


```
Get Started JS primeiroCodigo.js ×  
JS primeiroCodigo.js  
1> console.log('Olá mundo!'); 1°  
2> console.log('Olá mundo!'); 2°  
3> console.log('Olá mundo!'); 3°
```

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

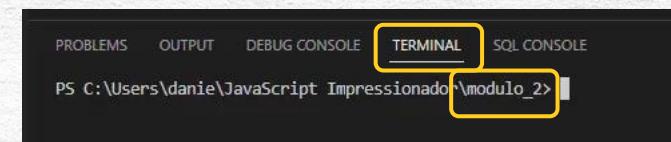
Instrução 1º

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.



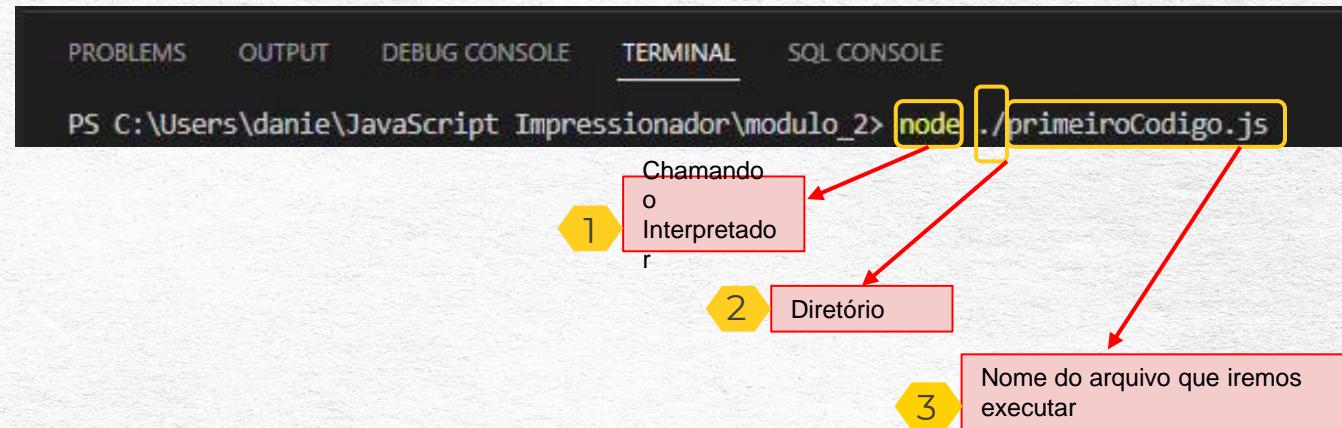
Instrução 2º

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 content:

```
Get Started JS primeiroCodigo.js X
JS primeiroCodigo.js
1   console.log('Eu acabei de escrever meu primeiro código JavaScript!!');

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL SQL CONSOLE
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**.

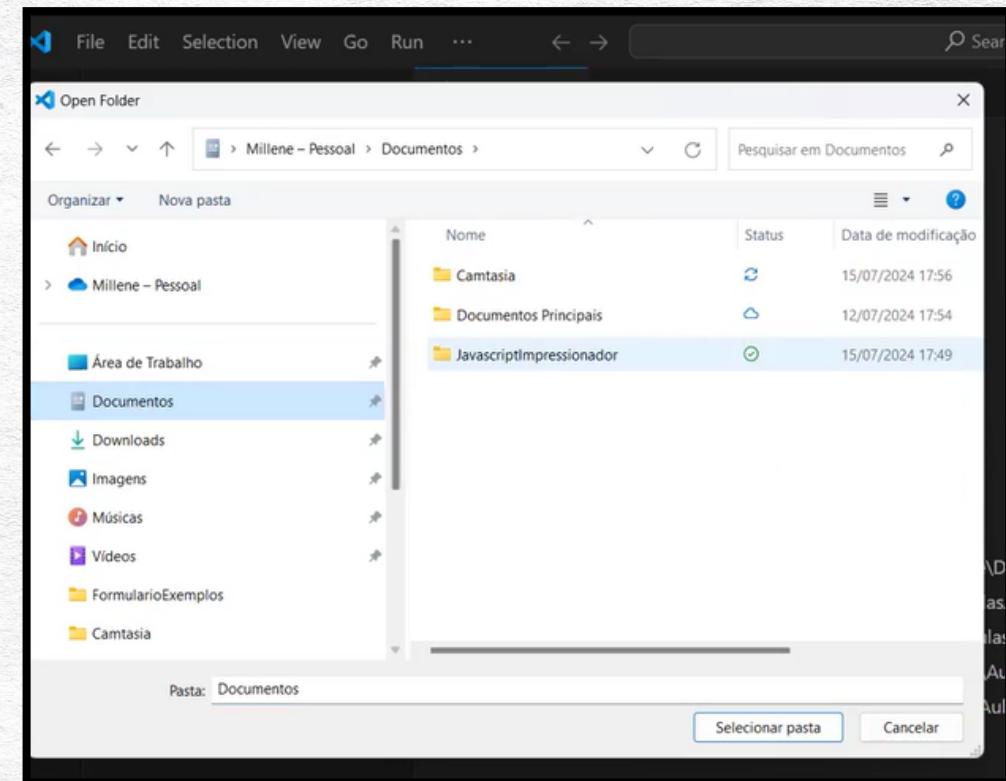
- **Verificar a Conexão:**

- 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
```

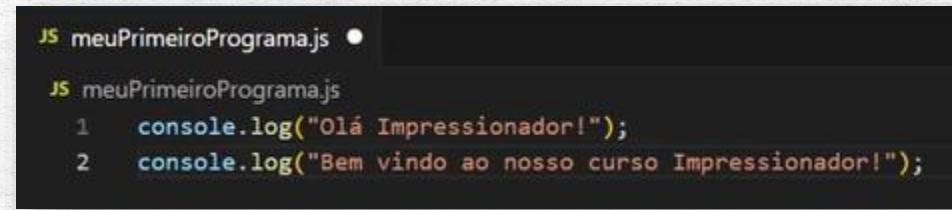
Erro Comum: Não Salvar Alterações no VS Code

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").

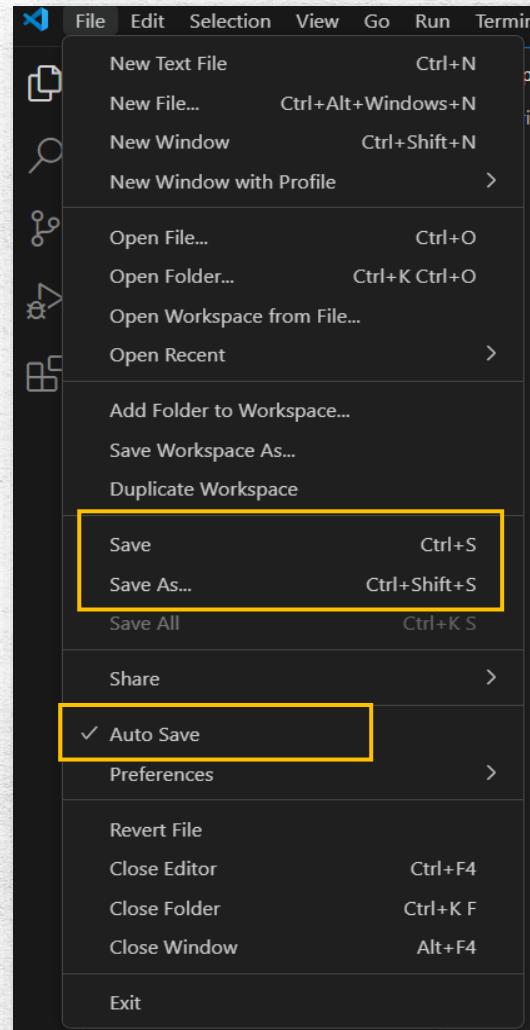
- **Habilite o Auto Save:**

- 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 two panes. The left pane displays a file tree under the directory 'JAVASCRIPTIMPRESSIONADOR'. It shows a folder 'novaPasta' containing a file 'script.js'. Below it is another file 'meuPrimeiroPrograma.js'. The right pane shows the contents of 'script.js':

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

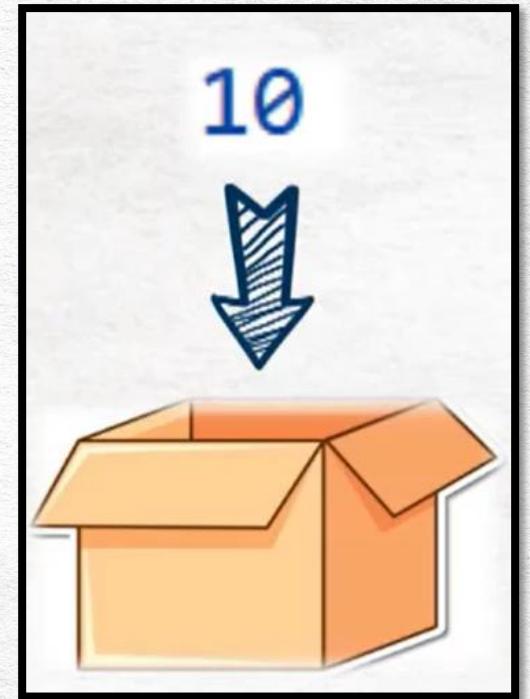
O que são Variáveis em JavaScript?

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.

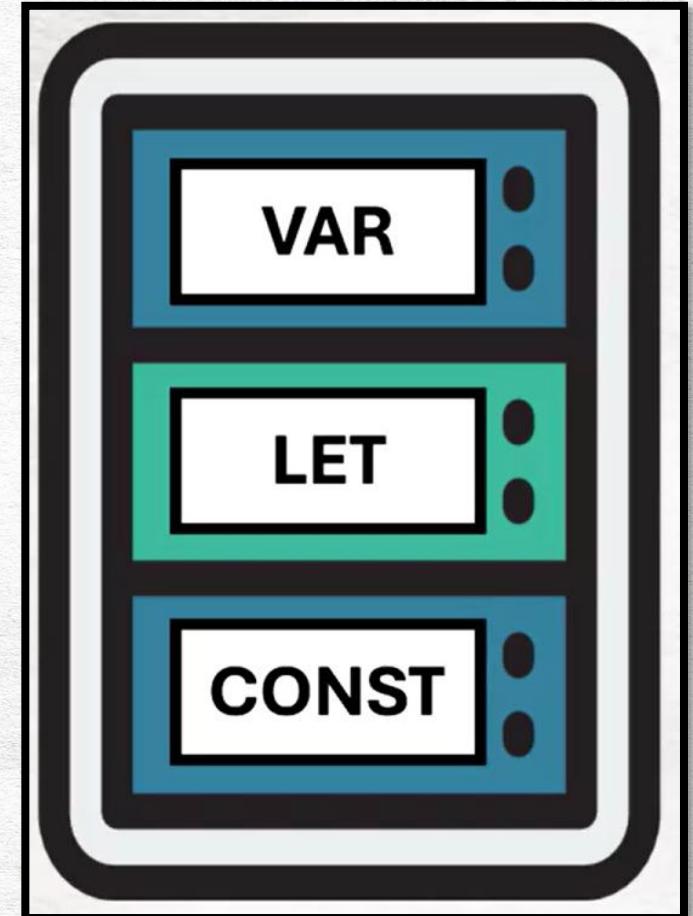
3. const (2015 em diante)

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 <code>undefined</code>	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.

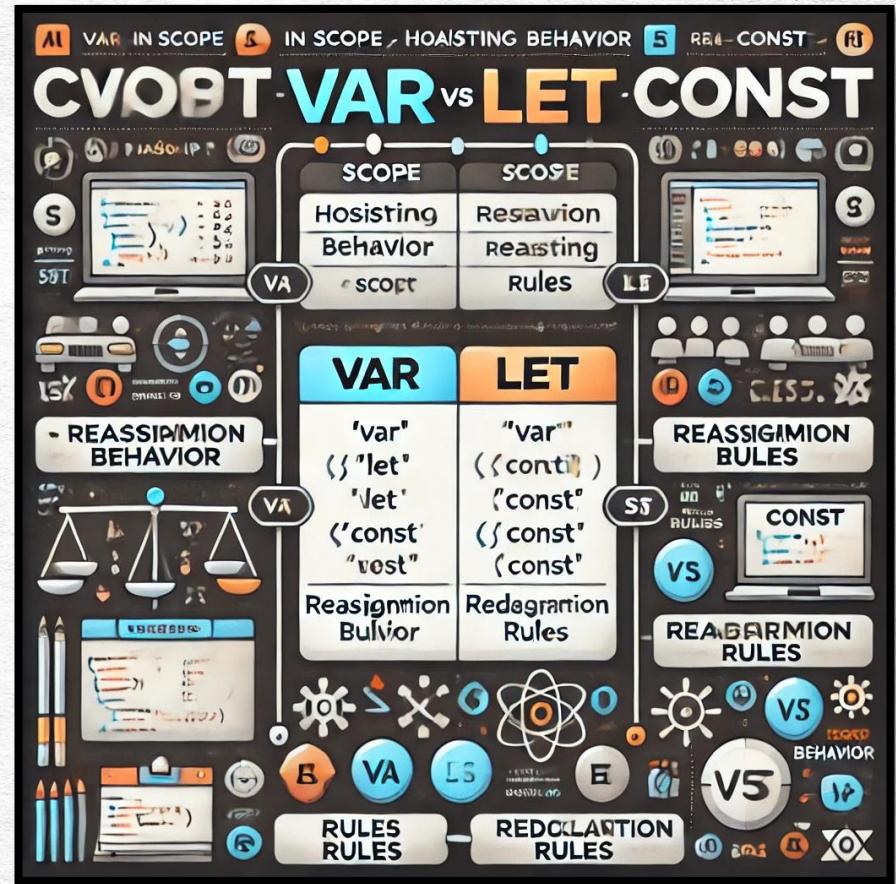
```
const nome = "Lucas";
const nome = "Carlos"; // Erro! Não é permitido redeclarar uma variável
```

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 **constantes**, utilize **letras 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.

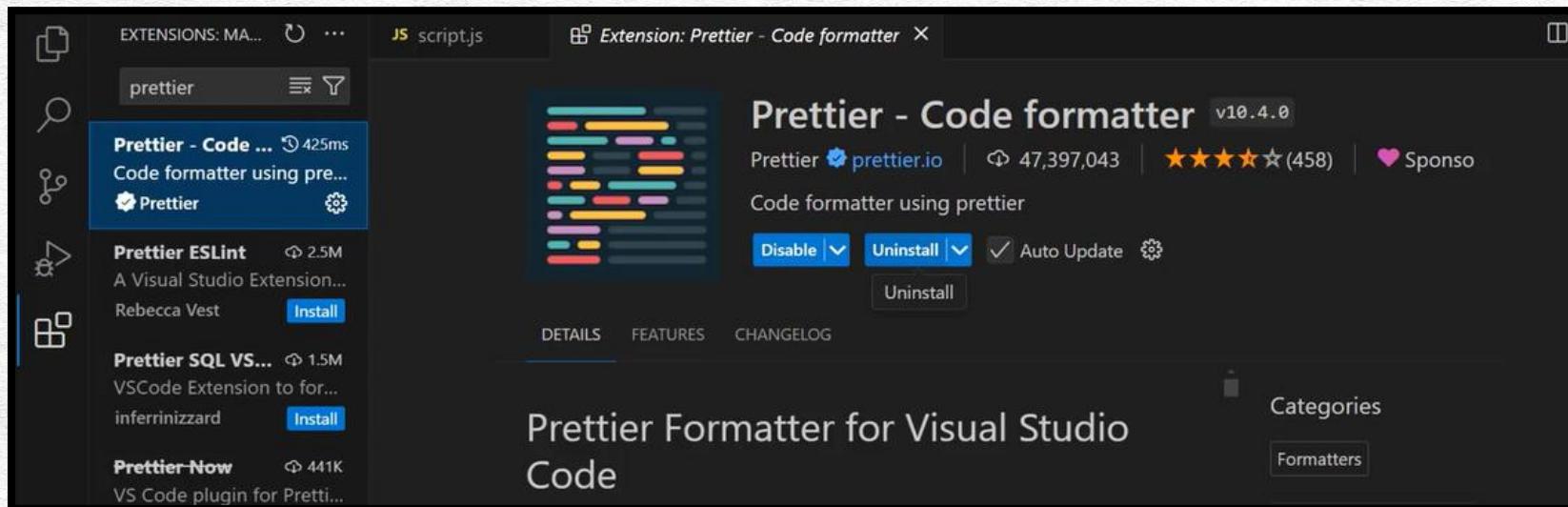
Adotar essas regras e boas práticas ao nomear variáveis e funções em seu código JavaScript é fundamental para criar projetos mais eficientes, claros e de fácil manutenção.

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 Comparaçāo 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?

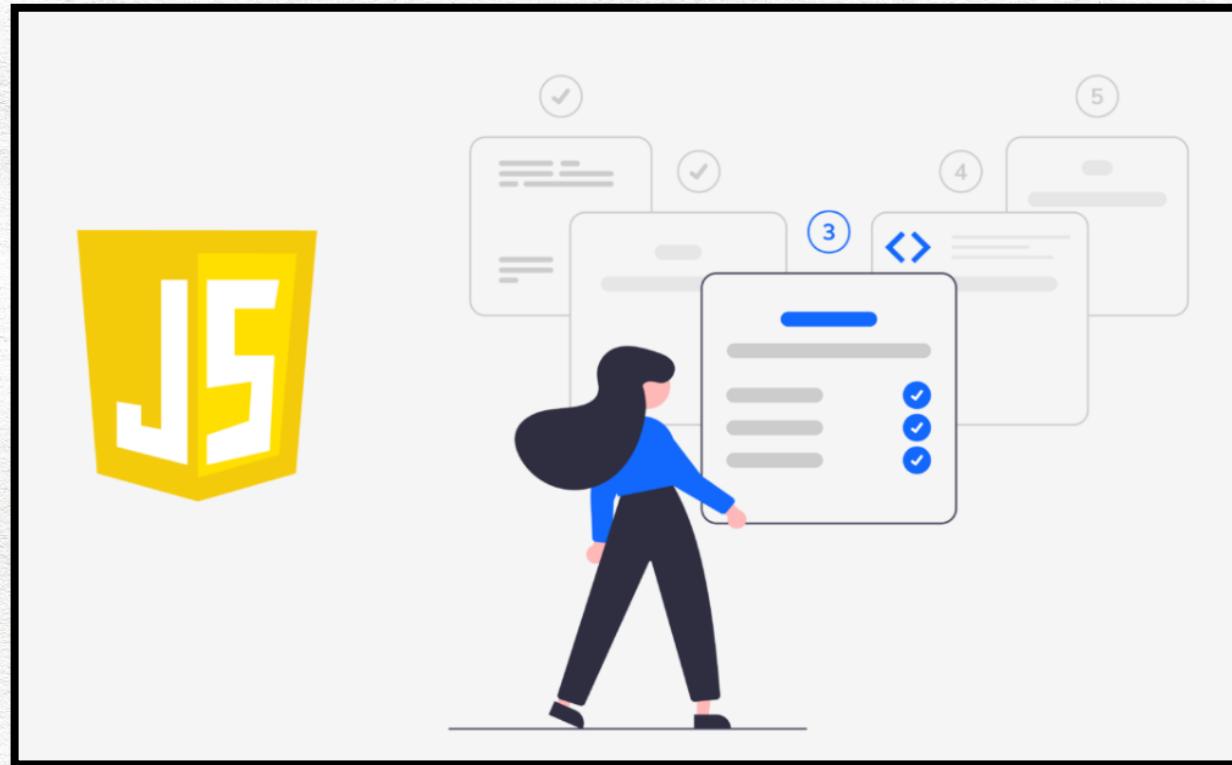
No JavaScript, o operador **==** realiza **coerção implícita** para determinar se dois valores são equivalentes. Segundo as regras do JavaScript definidas pela linguagem:

- **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[1], 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 armazena 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.

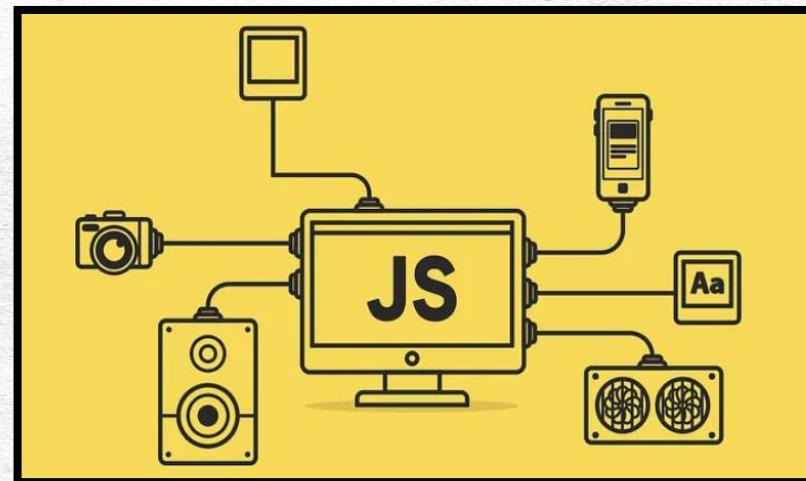
As funções são uma das principais características do JavaScript, permitindo que criemos códigos dinâmicos e reutilizáveis. Entre os conceitos mais poderosos estão as **funções de alta ordem**. Nesta aula, vamos entender o que elas são, como funcionam, e como aplicá-las no dia a dia da programação.

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
9     console.log(primeiraVariavelDoCodigo);
10    secondFunction();
11
12}
13
14 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 HTML, CSS, JAVASCRIPT, 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 'JS Reference' sections for 'JS by Category' and 'JS by Alphabet'. The main content area is titled 'JavaScript Array Methods and Properties'. It features a table with columns for 'Name' and 'Description'.

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()**

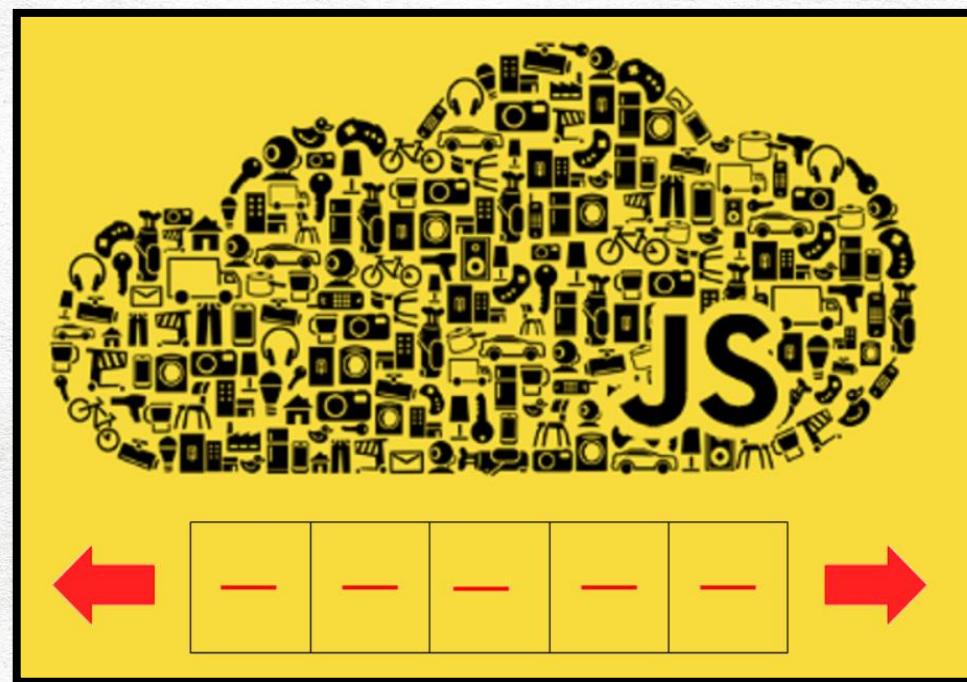
O método **shift()** remove o **primeiro elemento** de um array e o retorna, alterando o array original.

```
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(infinito)); // false
console.log(Number.isFinite(infinito2)); // 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.

Módulo 6

ESTRUTURAS CONDICIONAIS E DE REPETIÇÃO

ESTRUTURAS CONDICIONAIS E DE REPETIÇÃO

ESTRUTURAS CONDICIONAIS E DE REPETIÇÃO

Sabendo que o Javascript executa seu código de cima para baixo, um depois do outro, pode surgir a pergunta "Será que existe alguma maneira de mudarmos isso?". Vamos imaginar o seguinte cenário:

Vamos criar um site que será um portal de vendas de ingressos de futebol dos times Flamengo e Fluminense quando o jogo for no Estádio do Maracanã, para isso temos o seguinte código:

```
1 const mensagemDeBoasVindas1 = 'Bem vindo, torcedor Tricolor!';
2 const mensagemDeBoasVindas2 = 'Bem vindo, torcedor Rubro-Negro!';
3
4 console.log(mensagemDeBoasVindas1);
5 console.log(mensagemDeBoasVindas2);
6
7 const mensagemDeEscolha = 'Clique no setor para o qual deseja comprar seu ingresso!';
8 const mensagemFinal = 'Divirta-se no Maracanã!!!!';
9
10 console.log(mensagemDeEscolha);
11 console.log(mensagemFinal);
```

No código acima, declaramos as variáveis com mensagens armazenadas e `console.log()` para imprimirmos na tela essas mensagens.

Um dado que é muito relevante para nós seria qual o time que a pessoa torce, para imprimir a mensagem correta para o torcedor do Fluminense e torcedor do Flamengo. Então imagine que nós sabemos qual é o time do torcedor, que nesse momento será Fluminense.

Vamos analisar, faz sentido eu mostrar a mensagem 'Bem vindo, torcedor Rubro-Negro!' para esse torcedor?? Ou vice-e-versa?

Em um cenário mais amplo, imagine que esse Portal receberá inúmeros usuários diferentes e queremos que o nosso programa ao ser utilizado por um usurário não seja finalizado, e sim que esteja sempre pronto para ser utilizado por outro usuário logo em seguida.

É exatamente isso que iremos aprender com esse módulo! Vamos entender os conceitos de estruturas condicionais e de repetição.

Então para compreendermos melhor esse cenário que criamos o conceito de **Controle de Fluxo de Execução** tem que fazer parte do seu aprendizado.

Controles de fluxo são estruturas ou comandos que permitem desviar o **fluxo do programa**, ou seja, permitem que os desenvolvedores controlem o **fluxo de execução** de um programa com mais precisão. Controlar a **execução** de um programa é controlar através de um parâmetro, repetir um determinado processo X vezes, colocar alguma condição para que ocorra algo no código.

Um dado que é muito relevante para nós seria qual o time que a pessoa torce, para imprimir a mensagem correta para o torcedor do Fluminense e torcedor do Flamengo. Então imagine que nós sabemos qual é o time do torcedor, que nesse momento será Fluminense.

Vamos analisar, faz sentido eu mostrar a mensagem 'Bem vindo, torcedor Rubro-Negro!' para esse torcedor?? Ou vice-e-versa?

Em um cenário mais amplo, imagine que esse Portal receberá inúmeros usuários diferentes e queremos que o nosso programa ao ser utilizado por um usurário não seja finalizado, e sim que esteja sempre pronto para ser utilizado por outro usuário logo em seguida.

É exatamente isso que iremos aprender com esse módulo! Vamos entender os conceitos de estruturas condicionais e de repetição.

Então para compreendermos melhor esse cenário que criamos o conceito de **Controle de Fluxo de Execução** tem que fazer parte do seu aprendizado.

Controles de fluxo são estruturas ou comandos que permitem desviar o **fluxo do programa**, ou seja, permitem que os desenvolvedores controlam o **fluxo de execução** de um programa com mais precisão. Controlar a **execução** de um programa é controlar através de um parâmetro, repetir um determinado processo X vezes, colocar alguma condição para que ocorra algo no código.

ESTRUTURAS CONDICIONAIS

Servem para estabelecer blocos de código que apenas devem ser executados caso uma condição seja estabelecida, ou seja a estrutura condicional permite ao Javascript executar um trecho de código somente em determinadas condições.

As estruturas condicionais estão ligadas à tomada de decisão de um algoritmo. O tipo de valor que elas retornam serão do **tipo booleano**, ou seja, ao utilizar uma estrutura condicional ela irá retornar verdadeiro ou falso, e o algoritmo executará o bloco de comandos relativos a este resultado.



Será que ele leva um guarda-chuva ou não?

Dado o estado do céu, é melhor levar sim!



shutterstock.com - 1986844901

Estrutura condicional – IF

IF significa "Se" em inglês, o **IF** é a palavra chave que iremos empregar quando quisermos criar um bloco condicional.

A estrutura condicional **if** permite ao Javascript executar um trecho de código somente se uma determinada condição for verdadeira, ou seja, a condicional **if** executa a afirmação.

A estrutura básica do **if** em Javascript é :

```
if (Condição) {  
    Código que será  
    executado se a condição  
    for verdadeira  
}
```

Essa condição é feita por operadores lógicos como: **maior, menor, menor igual, igual, maior igual, diferente**. Também podem ser usados ou não em conjunto com: **e, ou**, etc.

No caso do nosso cenário do Portal do torcedor, para aplicarmos essa estrutura condicional utilizamos o if para criar uma condição de que SE o torcedor for de determinado tipo ele imprimi a mensagem correta ao torcedor. Vamos utilizar o exemplo do time Fluminense e imagine que nós já temos os dados do nosso usuário.

Criamos a condicional SE o torcedor for do Fluminense imprima a mensagem 'Bem vindo, torcedor do Tricolor!':

```
if (usuario.time === 'Fluminense') {  
  console.log(mensagemDeBoasVindas1);  
}
```

Aqui criamos um estrutura condicional, que SE (**if**) o time do usuário for Fluminense, então se essa condição for verdadeira, imprima a mensagem armazenada na variável mensagemDeBoasVindas1.

Observe o retorno de quando executamos o nosso código com a nossa condicional primeiro com o time do Fluminense e um retorno com outro time.

```
1 const usuario = { nome: 'Daniel', idade: 29, time: 'Fluminense' };
2 const mensagemDeBoasVindas1 = 'Bem vindo, torcedor Tricolor!';
3 const mensagemDeBoasVindas2 = 'Bem vindo, torcedor Rubro-Negro!';
4
5 if (usuario.time === 'Fluminense') {
6 | console.log(mensagemDeBoasVindas1);
7 }
8 console.log(mensagemDeBoasVindas2);
9
10 const mensagemDeEscolha =
11 | 'Clique no setor para o qual deseja comprar seu ingresso!';
12 const mensagemFinal = 'Divirta-se no Maracanã!!!';
13
14 console.log(mensagemDeEscolha);
15 console.log(mensagemFinal);
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL SQL CONSOLE: MESSAGES

```
C:\Program Files\nodejs\node.exe .\aula5_1\exampleCode.js
Bem vindo, torcedor Tricolor!
Bem vindo, torcedor Rubro-Negro!
Clique no setor para o qual deseja comprar seu ingresso!
Divirta-se no Maracanã!!!
```

```
1 const usuario = { nome: 'Daniel', idade: 29, time: 'Vasco' };
2 const mensagemDeBoasVindas1 = 'Bem vindo, torcedor Tricolor!';
3 const mensagemDeBoasVindas2 = 'Bem vindo, torcedor Rubro-Negro!';
4
5 if (usuario.time === 'Fluminense') {
6 | console.log(mensagemDeBoasVindas1);
7 }
8 console.log(mensagemDeBoasVindas2);
9
10 const mensagemDeEscolha =
11 | 'Clique no setor para o qual deseja comprar seu ingresso!';
12 const mensagemFinal = 'Divirta-se no Maracanã!!!';
13
14 console.log(mensagemDeEscolha);
15 console.log(mensagemFinal);
16
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL SQL CONSOLE: MESSAGES

```
C:\Program Files\nodejs\node.exe .\aula5_1\exampleCode.js
Bem vindo, torcedor Rubro-Negro!
Clique no setor para o qual deseja comprar seu ingresso!
Divirta-se no Maracanã!!!
```

```
1 const usuario = { nome: 'Daniel', idade: 29, time: 'Flamengo' };
2 const mensagemDeBoasVindas1 = 'Bem vindo, torcedor Tricolor!';
3 const mensagemDeBoasVindas2 = 'Bem vindo, torcedor Rubro-Negro!';
4
5 if (usuario.time === 'Fluminense') {
6   console.log(mensagemDeBoasVindas1);
7 }
8 if (usuario.time === 'Flamengo') {
9   console.log(mensagemDeBoasVindas2);
10}
11
12 const mensagemDeEscolha =
13   'Clique no setor para o qual deseja comprar seu ingresso!';
14 const mensagemFinal = 'Divirta-se no Maracanã!!!';
15
16 console.log(mensagemDeEscolha);
17 console.log(mensagemFinal);
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL SQL CONSOLE: MESSAGES

```
C:\Program Files\nodejs\node.exe .\aula5_1\exampleCode.js
Bem vindo, torcedor Rubro-Negro!
Clique no setor para o qual deseja comprar seu ingresso!
Divirta-se no Maracanã!!!
```

Repare que quando utilizamos o usuário for do time do Fluminense, a mensagem de 'Bem vindo Tricolor' aparece, mas também a mensagem do time do Flamengo também. E quando passamos o time Vasco ao usuário, o retorno não mostra a mensagem de 'Bem vindo Tricolor'.

Isso ocorre porque ele está atendendo apenas a **condição do IF**, se não for fluminense não irá imprimir a mensagem, e o programa continua a execução da próxima instrução que é o console.log() da mensagem de 'Bem Vindo Rubro-Negro'.

Um primeira solução para o nosso código seria criar uma estrutura condicional para a nossa segunda mensagem, aplicando a condição SE for do time do Flamengo imprima a 'mensagemDeBoasVindas2'.



ELSE

Else, em JavaScript, é uma versão reduzida da expressão **or else** do inglês, que significa "caso contrário", o **ELSE** virá sempre após do **IF**, definindo o comportamento que deve ser adotado caso a condição estabelecida não seja atendida, ou seja, ela é utilizada em conjunto com **if** quando vamos apresentar o bloco de comando que deve ser executado se a condição original do **if** não for verdadeira.

O **else** terá um **comportamento padrão** ou um conjunto de ações / instruções que **VÃO** ser executadas quando a condição do **if não for verdadeira**.

```
if (Condição ) { Verdadeiro = true
```

Código que será executado se a condição for verdadeira

```
} else { Falso = false
```

Código que será executado apenas quando a condição não for verdadeira

Módulo 6 – IF / ELSE IF / ELSE (2 / 7)

324

A **estrutura condicional IF /ELSE** será utilizado da seguinte maneira no nosso programa:

```
aula5_3 > JS exampleCode.js > ...
1 const usuario = { nome: 'Daniel', idade: 29, time: 'Fluminense' };
2 const mensagemDeBoasVindas1 = 'Bem vindo, torcedor Tricolor!';
3 const mensagemDeBoasVindas2 = 'Bem vindo, torcedor Rubro-Negro!';
4
5 if (usuario.time === 'Fluminense') {
6   console.log(mensagemDeBoasVindas1); ←
7 } else {
8   console.log(mensagemDeBoasVindas2);
9 }
10
11 const mensagemDeEscolha =
12   'Clique no setor para o qual deseja comprar seu ingresso!';
13 const mensagemFinal = 'Divirta-se no Maracanã!!!';
14
15 console.log(mensagemDeEscolha);
16 console.log(mensagemFinal);
17
18
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL SQL CONSOLE: MESSAGES Filter (e.g. text, !)

```
C:\Program Files\nodejs\node.exe .\aula5_1\exampleCode.js
Bem vindo, torcedor Tricolor!
Clique no setor para o qual deseja comprar seu ingresso!
Divirta-se no Maracanã!!!
```

```
1 const usuario = { nome: 'Daniel', idade: 29, time: 'Flamengo' };
2 const mensagemDeBoasVindas1 = 'Bem vindo, torcedor Tricolor!';
3 const mensagemDeBoasVindas2 = 'Bem vindo, torcedor Rubro-Negro!';
4
5 if (usuario.time === 'Fluminense') {
6   console.log(mensagemDeBoasVindas1);
7 } else { ←
8   console.log(mensagemDeBoasVindas2);
9 }
10
11 const mensagemDeEscolha =
12   'Clique no setor para o qual deseja comprar seu ingresso!';
13 const mensagemFinal = 'Divirta-se no Maracanã!!!';
14
15 console.log(mensagemDeEscolha);
16 console.log(mensagemFinal);
17
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL SQL CONSOLE: MESSAGES

```
C:\Program Files\nodejs\node.exe .\aula5_3\exampleCode.js
Bem vindo, torcedor Rubro-Negro!
Clique no setor para o qual deseja comprar seu ingresso!
Divirta-se no Maracanã!!!
```

Repare que ao passarmos os valores dos times, o seu retorno será a mensagem específica ao time do Usuário. No primeiro exemplo a condição é **VERDADEIRA**, então o bloco do if será executado, já no segundo, a condição do if **NÃO** é verdadeira e as instruções do else que serão aplicadas.



O comportamento que criamos utilizando dois IFs no Portal do Torcedor funciona, porém não é comum de ser utilizado, e o comportamento IF/ELSE só é possível quando temos apenas duas condições uma verdadeira e a outra que não é a verdadeira.

Vamos testar e verificar qual seria o retorno do IF/ELSE caso seja passado um time diferente:

```
1 const usuario = { nome: 'Daniel', idade: 29, time: 'Vasco' };
2 const mensagemDeBoasVindas1 = 'Bem vindo, torcedor Tricolor!';
3 const mensagemDeBoasVindas2 = 'Bem vindo, torcedor Rubro-Negro!';
4
5 if (usuario.time === 'Fluminense') {
6   console.log(mensagemDeBoasVindas1);
7 } else {
8   console.log(mensagemDeBoasVindas2);
9 }
10
11 const mensagemEscolha =
12   'Clique no setor para o qual deseja comprar seu ingresso!';
13 const mensagemFinal = 'Divirta-se no Maracanã!!!';
14
15 console.log(mensagemEscolha);
16 console.log(mensagemFinal);
17
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL SQL CONSOLE: MESSAGES

```
C:\Program Files\nodejs\node.exe .\aula5_3\exampleCode.js
Bem vindo, torcedor Rubro-Negro!
Clique no setor para o qual deseja comprar seu ingresso!
Divirta-se no Maracanã!!!
```

Observe que o time do nosso usuário nesse momento é o Vasco, mas o retorno que ele me dá é a informação da mensagem que deveria ser para torcedores do Flamengo.

Isso ocorre porque a estrutura condicional **ELSE** sempre será aplicado quando a condição do seu **IF** não for verdadeira, ou seja, eu poderia passar diferentes nomes de time que ele sempre apareceria a mensagem do Flamengo que está armazenada na variável 'mensagemDeBoasVindas2'.

No cenário anterior nosso Portal do Torcedor precisa ter um retorno diferente para os times diferentes e utilizar a **estrutura IF/ELSE** não funcionaria porque qualquer time que não fosse o Fluminense (verdadeiro na condição do if), teria a sua saída com as instruções do **ELSE** como padrão, ou seja, imprimíramos a mensagem do Flamengo para os demais torcedores, e isso seria um problema.

Nesse caso vamos conhecer mais uma estrutura condicional que podemos utilizar o **ELSE IF**.

IF / ELSE IF /ELSE

É utilizado quando queremos realizar múltiplas avaliações sucessivas. Podemos fazer uso da estrutura **if ... else ifelse**, ou seja, "SE...Caso contrário seCaso contrário". Iremos utilizar para especificar uma nova condição para testar, se a primeira condição não for verdadeira.

Se a condição do **if não for verdadeira**, ele entrará no bloco do **else if** e testará a condição dele, se ela for verdadeira a instrução do seu bloco será utilizado, caso contrário ele continuará e passará para o **else** retornar a instrução padrão que ele possui.

if (Condição 1) {

Código que será executado se a primeira condição for verdadeira

} else if (Condição 2) {

Código que será executado se a segunda condição for verdadeira

} else {

Código que será executado apenas quando nenhuma condição for verdadeira

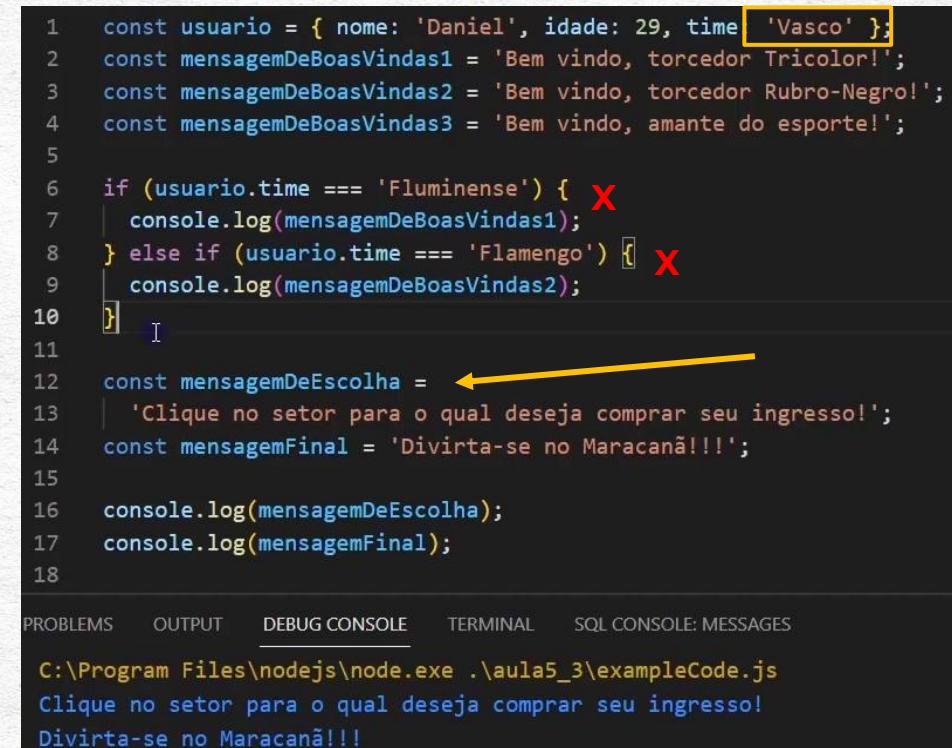
Você pode questionar o porquê utilizar o else no final da estrutura, já que estaremos criando condições diferentes para termos o retorno das mensagens corretas, mas repare o que acontece quando não adicionamos o else.

Podemos observar que o torcedor, nesse caso do Vasco, não foi contemplado por nenhuma das condições.

Quando o valor não é contemplado pela condição, ele pula e vai à próxima condição, continua não sendo verdadeira, pula e vai para próxima instrução que seria imprimir a mensagemDeEscolha.

Nesse caso é importante utilizar a estrutura else, para deixar um padrão caso nenhuma das condições seja contemplada:

```
if (usuario.time === 'Fluminense') {
  console.log(mensagemDeBoasVindas1);
} else if (usuario.time === 'Flamengo') {
  console.log(mensagemDeBoasVindas2);
} else {
  console.log(mensagemDeBoasVindas3);
}
```



```
1 const usuario = { nome: 'Daniel', idade: 29, time: 'Vasco' };
2 const mensagemDeBoasVindas1 = 'Bem vindo, torcedor Tricolor!';
3 const mensagemDeBoasVindas2 = 'Bem vindo, torcedor Rubro-Negro!';
4 const mensagemDeBoasVindas3 = 'Bem vindo, amante do esporte!';
5
6 if (usuario.time === 'Fluminense') { X
7   console.log(mensagemDeBoasVindas1);
8 } else if (usuario.time === 'Flamengo') { X
9   console.log(mensagemDeBoasVindas2);
10 }
11
12 const mensagemDeEscolha = ←
13   'Clique no setor para o qual deseja comprar seu ingresso!';
14 const mensagemFinal = 'Divirta-se no Maracanã!!!';
15
16 console.log(mensagemDeEscolha);
17 console.log(mensagemFinal);
18
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL SQL CONSOLE: MESSAGES

C:\Program Files\nodejs\node.exe .\aula5_3\exampleCode.js

Clique no setor para o qual deseja comprar seu ingresso!

Divirta-se no Maracanã!!!

Vamos criar uma estrutura condicional agora para que o nosso Portal também imprima uma mensagem específica para o nosso torcedor do Vasco, para isso podemos concatenar ELSE IF, e criar novas condições.

```
1  const usuario = { nome: 'Daniel', idade: 29, time: 'Vasco' };
2  const mensagemDeBoasVindas1 = 'Bem vindo, torcedor Tricolor!';
3  const mensagemDeBoasVindas2 = 'Bem vindo, torcedor Rubro-Negro!';
4  const mensagemDeBoasVindas3 = 'Bem vindo, torcedor Vascaíno!';
5  const mensagemDeBoasVindas4 = 'Bem vindo, amante do esporte!';

6
7  if (usuario.time === 'Fluminense') {
8    console.log(mensagemDeBoasVindas1);
9  } else if (usuario.time === 'Flamengo') {
10   console.log(mensagemDeBoasVindas2);
11 } else if (usuario.time === 'Vasco') { ←
12   console.log(mensagemDeBoasVindas3);
13 } else {
14   console.log(mensagemDeBoasVindas4);
15 }

16
17 const mensagemDeEscolha =
18   'Clique no setor para o qual deseja comprar seu ingresso!';

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL SQL CONSOLE: MESSAGES
C:\Program Files\nodejs\node.exe .\aula5_3\exampleCode.js
Bem vindo, torcedor Vascaíno! ←
Clique no setor para o qual deseja comprar seu ingresso!
Divirta-se no Maracanã!!!
```

Observe que o nosso programa funciona e agora contempla também o time do Vasco e mantemos o retorno padrão do Else para caso seja colocado um valor diferente de Fluminense, Flamengo e Vasco.



Outro questionamento que pode ser levantado é qual seria a diferença de criar dois IFs e utilizar a estrutura IF ...ELSE IF.

```
if (usuario.time === 'Fluminense') {  
    console.log(mensagemDeBoasVindas1);  
}  
if (usuario.time === 'Flamengo') {  
    console.log(mensagemDeBoasVindas2);  
}
```

```
if (usuario.time === 'Fluminense') {  
    console.log(mensagemDeBoasVindas1);  
} else if (usuario.time === 'Flamengo') {  
    console.log(mensagemDeBoasVindas2);  
}
```

A diferença é que quando você executar o programa, no exemplo 1 o Javascript irá executar uma condição IF e depois tentará executar a outra condição IF.

No exemplo 2, o Javascript executará a condição IF, SE ela for verdadeira a estrutura condicional é parada e o bloco de instrução dentro do IF será executado, apenas SE a condição NÃO for verdadeira, ele executará a condição ELSE IF.

Agora imagine o cenário onde nosso Portal do Torcedor não tenha apenas 2 times, mais sim 8 times, utilizar a estrutura **IF/ELSE** não funcionaria e já entendemos que é uma estrutura para quando temos apenas uma condição verdadeira ou não, com várias condições diferentes ele não consegue ser aplicado.

Tentar utilizar **vários IFs** faz o **código** ser **mais lento**, pois ele tentaria executar cada if até encontrar a condição, e mesmo encontrando ele continuar a executar os IFs até terminar a lista.

Utilizar a estrutura **IF/ELSE IF/ELSE** um após o outro funcionaria mas deixa o nosso **código VERBOSO**, ou seja, não é legível e fácil de compreender.

```
1 const usuario = { nome: 'Daniel', idade: 29, time: 'Fluminense' };
2 const mensagemDeBoasVindas1 = 'Bem vindo, torcedor Tricolor!';
3 const mensagemDeBoasVindas2 = 'Bem vindo, torcedor Rubro-Negro!';
4 const mensagemDeBoasVindas3 = 'Bem vindo, torcedor Vascaíno!';
5 const mensagemDeBoasVindas4 = 'Bem vindo, torcedor Santista!';
6 const mensagemDeBoasVindas5 = 'Bem vindo, torcedor do Náutico!';
7 const mensagemDeBoasVindas6 = 'Bem vindo, torcedor do Atlético Mineiro!';
8 const mensagemDeBoasVindas7 = 'Bem vindo, torcedor do Cruzeiro!';
9 const mensagemDeBoasVindas8 = 'Bem vindo, torcedor do Corinthians!';
```

```
13 if (usuario.time === 'Fluminense') {
14   console.log(mensagemDeBoasVindas1);
15 } else if (usuario.time === 'Flamengo') {
16   console.log(mensagemDeBoasVindas2);
17 } else if (usuario.time === 'Vasco') {
18   console.log(mensagemDeBoasVindas3);
19 } else if (usuario.time === 'Santos') {
20   console.log(mensagemDeBoasVindas2);
21 } else if (usuario.time === 'Náutico') {
22   console.log(mensagemDeBoasVindas3);
23 } else if (usuario.time === 'AtléticoMG') {
24   console.log(mensagemDeBoasVindas2);
25 } else if (usuario.time === 'Cruzeiro') {
26   console.log(mensagemDeBoasVindas3);
27 } else if (usuario.time === 'Corinthians') {
28   console.log(mensagemDeBoasVindas3);
```

Então sempre optaremos por construir um código com boas práticas, legível e o mais simples / fácil possível de se compreender e entender.

Para isso vamos aprender a estrutura condicional SWITCH / CASE.

SWITCH / CASE

O switch case é uma estrutura condicional do Javascript como o if, e serve para analisar os valores e executar um bloco de código condicionalmente.

Em sua sintaxe ela precisa de uma **condição ou expressão (usuario.time + usuario.nome)**.

```
switch (usuario.time) {
```

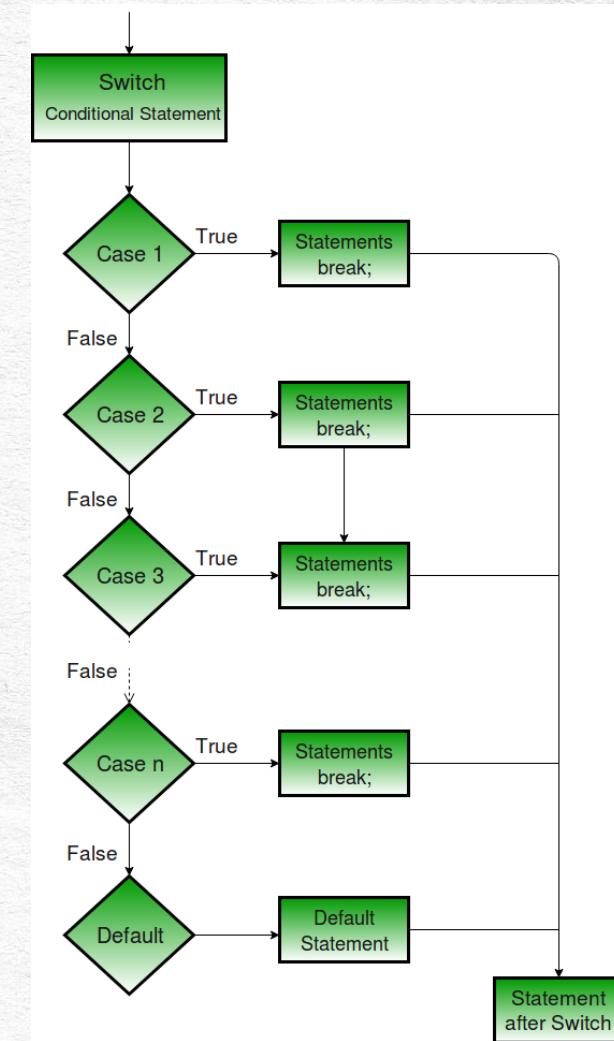
```
switch [usuario.time]+ usuario.nome) {
```

O **SWITCH** primeiro ele avalia essa condição e depois irá verificar nos casos (**CASE**) para encontrar um possível verdadeiro.

Quando entramos em um caso e ele for verdadeira, iremos aplicar a instrução do bloco e precisamos utilizar o **BREAK** para paramos de procurar novos casos, pois já conseguimos um retorno verdadeiro aquela condição.

E ao final do nosso Switch, precisamos também colocar um **caso PADRÃO**, que funciona como a estrutura ELSE, caso nenhum caso for verdadeiro aplique essa instrução padrão.

E para isso utilizamos o bloco **DEFAULT**.



Então vamos aplicar essa nova estrutura e observar nosso código seguindo as condições de times e mensagens que criamos.

```
13 switch (usuario.time) {  
14     case 'Fluminense':  
15         console.log(mensagemDeBoasVindas1);  
16         break;  
17     case 'Flamengo':  
18         console.log(mensagemDeBoasVindas2);  
19         break;  
20     case 'Vasco':  
21         console.log(mensagemDeBoasVindas3);  
22         break;  
23     case 'Náutico':  
24         console.log(mensagemDeBoasVindas4);  
25         break;  
26     case 'AtléticoMG':  
27         console.log(mensagemDeBoasVindas5);  
28         break;
```

No código ao lado, começamos a estutarar nossos casos.

O **SWITCH** recebe a condição, que nesse caso é verificar o time do usuário. Dentro do seu **bloco switch {}**, aplicamos os **CASE** que vão verificar se a informação é uma condição verdadeira ou não.

A sintaxe do **CASE** é :

```
case 'Informação':  
    Bloco de instrução;  
    break;
```

Caso a informação não for verdadeira, adicionamos mais um case (caso) para ser testado.

Uma das diferenças do **SWITCH** é caso algum **CASE** sejam verdadeiro aquela condição, ele irá aplicar diretamente ele.

Diferente do **ELSE IF** que vai procurando de estrutura em estrutura, o Switch é mais veloz, então você pode ter 20 casos, se um for verdadeiro, ele já o executará.

Mas e se ele não encontrar nenhum caso que seja verdadeiro aquela condição? O switch como explicamos anteriormente também possui um bloco padrão para quando a condição não for contemplada por nenhum case, que é chamado de **DEFAULT**.

A sintaxe do **DEFAULT** sempre será o último caso a ser escrito no bloco Switch:

PADRÃO	Default : Bloco de instrução; break; OPCIONAL
--------	---

```

28   case 'Náutico':
29     console.log(mensagemDeBoasVindas5);
30     break;
31   case 'AtléticoMG':
32     console.log(mensagemDeBoasVindas6);
33     break;
34   case 'Cruzeiro':
35     console.log(mensagemDeBoasVindas7);
36     break;
37   case 'Corinthians':
38     console.log(mensagemDeBoasVindas8);
39     break;
40   default: arrow pointing here
41     console.log(mensagemDeBoasVindasGeral);
42   } break; OPCIONAL

```

A palavra-chave **BREAK**, vem do inglês Parar / Interromper / Brecar. Todo **CASE** precisa de um **BREAK**, no caso do **DEFAULT** ele é **opcional**, já que o Default é a última instrução do Switch.

O **BREAK** tem a funcionalidade de parar uma execução e não permitir que o efeito cascata ocorra.

O **efeito cascata** é quando tudo começa a ser lido / executado de cima para baixo sem nenhum tipo de restrição que faça ele interromper.

No caso do Switch, quando retiramos o break, indicamos que queremos que ele passe de case em case , executando seus blocos de instrução.

Então podemos entender que sem o break, nosso código será lido assim:

caso - executa bloco – caso – executa bloco –caso –executa bloco....

Ele não tem uma parada quando a condição é contemplada.

```
13 switch (usuario.time) {  
14     case 'Fluminense':  
15         console.log(mensagemDeBoasVindas1);  
16     case 'Flamengo':  
17         console.log(mensagemDeBoasVindas2);  
18         break;  
19     case 'Vasco':  
20         console.log(mensagemDeBoasVindas3);  
21         break;  
22     case 'Santos':  
23         console.log(mensagemDeBoasVindas4);  
24         break;  
25     case 'Náutico':
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL SQL CONSOLE: MESSAGES

```
C:\Program Files\nodejs\node.exe .\aula5_4\exampleCode.js  
Bem vindo, torcedor Tricolor!  
Bem vindo, torcedor Rubro-Negro!  
Clique no setor para o qual deseja comprar seu ingresso!  
Divirta-se no Maracanã!!!
```

O **Switch** também me permite unir vários casos que possuem a mesmo bloco de informação, para não repetirmos os blocos, o **Switch aceita CASE consecutivos** e depois o nosso bloco de instrução.

Isso ocorre porque o **Switch** irá ler o **Case** e aplicará a instrução de baixo, como também é um **Case**, ela lerá e continuará realizando as instruções até encontrar um **Break** que interrompa o seu processo.

Imagine o cenário dos torcedores do Fluminense, São Paulo e Grêmio, todos esses times são chamado de 'Tricolor', então ao invés de criar uma Case para cada um , é possível uni-los para executar uma única instrução caso a condição seja contemplada.

```

1 const usuario = { nome: 'Daniel', idade: 29, time: 'Grêmio' };  

2 const mensagemDeBoasVindas1 = 'Bem vindo, torcedor Tricolor!';  

3 const mensagemDeBoasVindas2 = 'Bem vindo, torcedor Rubro-Negro!';  

4 const mensagemDeBoasVindas3 = 'Bem vindo, torcedor Vascaíno!';  

5 const mensagemDeBoasVindas4 = 'Bem vindo, torcedor Santista!';  

6 const mensagemDeBoasVindas5 = 'Bem vindo, torcedor do Náutico!';  

7 const mensagemDeBoasVindas6 = 'Bem vindo, torcedor do Atlético Mineiro!';  

8 const mensagemDeBoasVindas7 = 'Bem vindo, torcedor do Cruzeiro!';  

9 const mensagemDeBoasVindas8 = 'Bem vindo, torcedor do Corinthians!';  

10  

11 const mensagemDeBoasVindasGeral = 'Bem vindo, amante do esporte!';  

12  

13 switch (usuario.time) {  

14   case 'Fluminense':  

15   case 'São Paulo':  

16   case 'Grêmio':  

17     console.log(mensagemDeBoasVindas1);  

18     break;

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL SQL CONSOLE: MESSAGES

```
C:\Program Files\nodejs\node.exe .\aula5_4\exampleCode.js
Bem vindo, torcedor Tricolor!
Clique no setor para o qual deseja comprar seu ingresso!
Divirta-se no Maracanã!!!
```

```

13 switch (usuario.time) {  

14   case 'Fluminense':  

15   case 'São Paulo':  

16   case 'Grêmio':  

17     console.log(mensagemDeBoasVindas1);  

18     break;

```



```
switch (expressão) {  
    case 'valor1':  
        código que será  
        executado caso  
        expressão === 'valor1'  
        break;  
  
    case 'valor2':  
        código que será  
        executado caso  
        expressão === 'valor2'  
        break;  
    case 'valor3':  
    case 'valor4':  
        Instruções associadas aos valores 3  
        e 4  
        break;  
  
    default:  
        comportamento executado quando  
        nenhuma condição anterior satisfez o  
        valor avaliado  
}
```

SWITCH CASE – RESUMO E SINTAXE

O switch é uma estrutura para múltiplas avaliações, mas o nosso universo de casos possíveis é limitado e bem definido (pense em uma questão de múltipla escolha), o Switch – Case apresenta essa estrutura:



Voltando ao nosso exemplo do IF /ELSE que avaliava apenas dois times, nós vamos ver uma estrutura alternativa para o IF /ELSE, ou seja, para casos que você tenha apenas se a condição será verdadeira ou não.

```
1 const usuario = { nome: 'Daniel', idade: 29, time: 'Fluminense' };
2 const mensagemDeBoasVindas1 = 'Bem vindo, torcedor Tricolor!';
3 const mensagemDeBoasVindas2 = 'Bem vindo, torcedor Rubro-Negro!';
4
5 if (usuario.time === 'Fluminense') {
6   console.log(mensagemDeBoasVindas1);
7 } else {
8   console.log(mensagemDeBoasVindas2);
9 }
```

Então vamos utilizar essa código para aplicar uma estrutura alternativa e muito bacana para casos como esse.

Então relembrando temos a mensagem para o torcedor do Fluminense e Flamengo e estamos verificando **SE** ele for Fluminense imprima a mensagem 1, **Caso Contrário**, imprima a mensagem 2.

Uma das preocupações que sempre devemos ter é com a LEGIBILIDADE do nosso código, então toda vez que conseguirmos expressar a mesma coisa de uma forma mais simples, sucinta, poluindo menos o código e o tornando mais legível, isso é muito importante e uma ótima boa prática como desenvolvedor.

OPERADOR TERNÁRIO

condição booleana

? execute caso
verdadeiro: execute caso
falso

O **OPERADOR Condicional (TERNÁRIO)** é o único operador JavaScript que possui três operandos.

O **OPERADOR TERNÁRIO** recebe três argumentos: O primeiro é um **argumento de comparação**. O segundo é o resultado caso a comparação seja **verdadeira**. O terceiro é o resultado caso ela seja **falsa**.

Este operador é frequentemente usado como um atalho para a instrução if , ou seja, se a situação com a qual lidamos exigir uma simples estrutura de **IF / ELSE**, podemos também optar pelo operador ternário para representar o cenário

IF / ELSE

```
if (usuario.time === 'Fluminense') {  
    console.log(mensagemDeBoasVindas1);  
} else {  
    console.log(mensagemDeBoasVindas2);  
}
```

OPERADOR TERNÁRIO

```
usuario.time === 'Fluminense' ? console.log(mensagemDeBoasVindas1) : console.log(mensagemDeBoasVindas2);
```

No código acima mudaremos a Estrutura simples do IF / ELSE em um Operador Condicional Ternário.

A estrutura do Operador Ternário nos informa a condição que iremos verificar (**usuario.time === 'Fluminense'**) , fazendo a **pergunta**, essa condição é **verdadeira** (**?**) , se for verdadeira irá imprimir (`console.log(mensagemDeBoasVindas1)`), **caso contrário** (**:**) imprima (`console.log(mensagemDeBoasVindas2)`).

IF / ELSE IF / ELSE

```
if (usuario.time === 'Fluminense') {  
    console.log(mensagemDeBoasVindas1);  
} else if (usuario.time === 'Flamengo') {  
    console.log(mensagemDeBoasVindas2);  
} else if (usuario.time === 'Vasco') {  
    console.log(mensagemDeBoasVindas3);  
} else {  
    console.log(mensagemDeBoasVindas4);  
}
```

OPERADOR TERNÁRIO

```
usuario.time === 'Fluminense'  
? console.log(mensagemDeBoasVindas1)  
: usuario.time === 'Flamengo'  
? console.log(mensagemDeBoasVindas2)  
: usuario.time === 'Vasco'  
? console.log(mensagemDeBoasVindas3)  
: console.log(mensagemDeBoasVindasGeral);
```

Também é possível utilizar em estruturas do IF / ELSE IF / ELSE em um Operador Condicional Ternário.

A estrutura do Operador Ternário adicionaria cada condição que queremos aplicar um bloco de instrução. Isso é possível porque conseguimos encaixar operadores ternários.

```
7  usuario.time === 'Fluminense'  
8    ? console.log(mensagemDeBoasVindas1)  
9    : usuario.time === 'Flamengo'  
10   ? console.log(mensagemDeBoasVindas2)  
11   : usuario.time === 'Vasco'  
12   ? console.log(mensagemDeBoasVindas3)  
13   : usuario.time === 'Botafogo'  
14   ? console.log(mensagemDeBoasVindas3)  
15   : usuario.time === 'Santos'  
16   ? console.log(mensagemDeBoasVindas3)  
17   : usuario.time === 'Grêmio'  
18   ? console.log(mensagemDeBoasVindas3)  
19   : usuario.time === 'Internacional'  
20   ? console.log(mensagemDeBoasVindas3)  
21   : usuario.time === 'Cruzeiro'  
22   ? console.log(mensagemDeBoasVindas3)  
23   : console.log(mensagemDeBoasVindasGeral);
```

Porém assim como no outro cenário do IF /ELSE IF /ELSE, a conexão de vários Operadores Ternários, também faz com que o nosso código perca a sua LEGIBILIDADE e fugimos das boas práticas.

Reflita sempre que **nem sempre o que FUNCIONA está CORRETO.**

Então aplicar as boas práticas como programador é essencial para sua vida profissional como desenvolvedor.

Condições Booleanas são condições que entregam os valores de verdadeiro (true) ou falso (false), no caso das estruturas condicionais sempre esperamos que seu retorno seja um booleano, mesmo que você não esteja utilizando uma variável boolean.

CONDIÇÃO BOLENA

```
if (usuario.time === 'Fluminense') {  
  console.log(mensagemDeBoasVindas1);  
}
```

Essas condições se aplicam a todas as estruturas condicionais que vimos até agora, e também conseguimos aumentar a complexidade utilizando **expressões booleanas**, que são união de condições que também retornarão um valor booleano.

EXPRESSÃO BOOLEANA

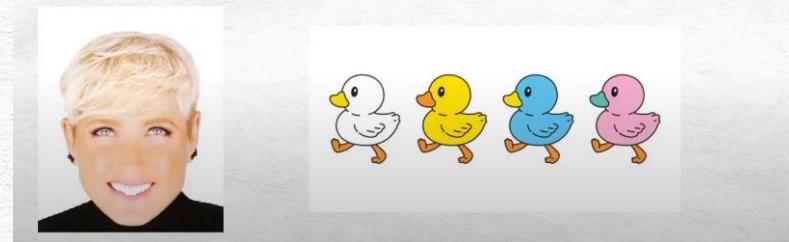
```
usuario.time === 'Fluminense' && usuario.idade >= 18
```

ESTRUTURAS DE REPETIÇÃO

Servem para estabelecer blocos de código que serão executados várias vezes, até que sua "condição de repetição" deixe de ser contemplada. Elas irão alterar o nosso controle de fluxo.

Um exemplo muito engraçado para entendermos o que é uma estrutura de repetição é a seguinte situação:

- “Cinco patinhos foram passear / Além das montanhas /Para brincar /A mamã gritou: Quá, quá, quá, quá/ Mas só quatro patinhos voltaram de lá.
- Quatro patinhos foram passear /Além das montanhas /Para brincar ...”



Lembra da música dos patinhos? Ela é um exemplo claro, que temos uma condição (quando a mãe chama eles tem que voltar) , nesse momento retornando uma informação (menos patos do que foram) e precisamos começar de novo para aplicar a condição, retorna mais um patinho e repetir novamente a canção.

Porém quando a condição não for mais contemplada, ou seja, não ter mais patinhos para voltar, o nosso laço de repetição para de acontecer.

Resumindo temos uma estrutura de repetição acontecendo, com uma informação que está alterando vez após vez (**iteração** após iteração) até que sua condição não seja contemplada e pare de executar.

Em JavaScript, um loop é uma estrutura de controle que permite executar um bloco de código repetidamente enquanto uma condição específica for verdadeira. Isso é fundamental para automatizar tarefas repetitivas e processar coleções de dados de forma eficiente.

Existem várias formas de loops em JavaScript, incluindo o loop "for", o loop "while", o loop "do-while" e outros. Cada tipo de loop tem sua própria sintaxe e uso adequado, mas todos seguem o mesmo princípio básico: repetir a execução de um bloco de código até que uma condição seja atendida.

Ao usar um loop, você geralmente define uma condição de término que determina quando o loop deve parar de executar. Enquanto essa condição for verdadeira, o bloco de código dentro do loop é executado repetidamente. Assim que a condição se torna falsa, o loop é encerrado e o controle do programa passa para o código que vem após o loop.

Funcionalidades importantes

- **Automatização de Tarefas Repetitivas:** Os loops são úteis para executar tarefas repetitivas de forma automática. Por exemplo, se você precisa executar uma operação em uma lista de itens, não é necessário repetir o código para cada item individualmente. Você pode usar um loop para percorrer a lista e executar a operação em cada item.
- **Flexibilidade:** Os loops permitem lidar com diferentes situações de forma flexível. Você pode ajustar as condições do loop ou o comportamento dentro do loop conforme necessário para atender aos requisitos específicos do seu código.

- **Eficiência:** Utilizar loops pode tornar seu código mais eficiente, especialmente em casos em que você precisa lidar com grandes quantidades de dados. Em vez de escrever código repetitivo manualmente, você pode usar um loop para processar os dados de forma mais rápida e eficiente.
- **Condições de Término:** Os loops são executados até que uma condição específica seja atendida. Isso pode ser uma condição simples, como percorrer uma lista até o final, ou uma condição mais complexa, como parar o loop quando um determinado valor for encontrado.
- **Controle de Iteração - "continue" e "break":** Dentro de um loop, você tem controle sobre o fluxo de execução usando as instruções "continue" e "break". A instrução "continue" interrompe a iteração atual do loop e passa para a próxima iteração, enquanto a instrução "break" interrompe completamente a execução do loop.
- **Aninhamento de Loops:** Você pode colocar um loop dentro de outro loop, o que é chamado de aninhamento de loops. Isso é útil quando você precisa lidar com estruturas de dados multidimensionais ou executar tarefas que exigem iterações múltiplas.

Essas funcionalidades são fundamentais para entender como usar loops de forma eficaz em programação, permitindo automatizar tarefas repetitivas, escrever código mais flexível e eficiente e controlar o fluxo de execução do programa.

Nesse momento iremos aprender a primeira estrutura de repetição, o **FOR**.

FOR

Para utilizar o for, usamos a palavra-chave **for** seguida de três declarações, primeiramente iniciamos a variável que será o controlador do nosso laço de repetição, logo em seguida precisamos informar a condição a ser atendida e, por último, uma expressão que será executado ao final de cada iteração do for, normalmente utilizamos para incrementar a variável que será utilizada como controlador do nosso laço.

- 1 – Uma declaração inicial, que será estabelecida quando a execução do código chegar pela primeira vez na estrutura de repetição.
 - 2 – A condição para que o bloco de instruções continue sendo repetido.
 - 3 – Uma operação que será realizada ao fim de cada iteração.

```
for (declaração ; condição; operação pós) {  
    inicial                                iteração
```

Código que será executado enquanto a condição for verdadeira

}

O loop **for** no JavaScript é uma estrutura de controle que permite executar um bloco de código repetidamente, enquanto uma condição específica for verdadeira. Aqui está a sintaxe básica:

```
for (inicialização; condição; expressão_final) {  
    // bloco de código a ser repetido  
}
```

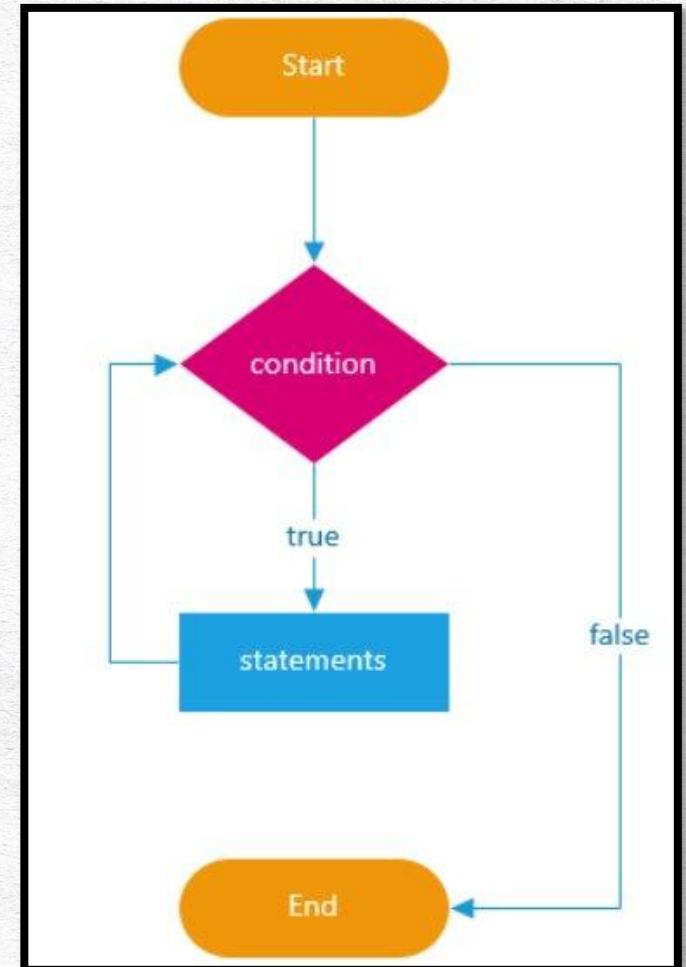
O **for** recebe 3 declarações, iniciamos o nosso contador, indicamos a condição da estrutura e por fim incrementamos o nosso contador ao final da iteração.

- **inicialização**: É onde você inicializa o contador do loop. Normalmente, você define uma variável que será usada como contador.
- **condição**: É a expressão que é avaliada a cada iteração do loop. Se for verdadeira, o bloco de código dentro do loop é executado. Se for falsa, o loop é encerrado.
- **incremento**: É onde você atualiza o contador do loop. Normalmente, você incrementa ou decrementa o contador.

Vamos explicar o fluxo do **for** usando os nomes dos argumentos/processos:

- **Inicialização:** No início do loop **for**, a inicialização é executada. Neste estágio, uma variável de controle (geralmente chamada de **i**) é inicializada com um valor inicial.
- **Condição:** Após a inicialização, a condição é avaliada. Se a condição for verdadeira, o bloco de código dentro do loop é executado. Se a condição for falsa, o loop é encerrado.
- **Bloco de código:** Se a condição for verdadeira, o bloco de código dentro do loop é executado. Este bloco pode conter qualquer código que você queira repetir várias vezes.
- **Incremento/Decremento:** Após a execução do bloco de código, o incremento ou decremento é aplicado à variável de controle. Este passo é usado para alterar o valor da variável de controle e, portanto, controlar a execução do loop.
- **Volta à Condição:** Depois que o incremento/decremento é aplicado, o loop volta à etapa de avaliação da condição. Se a condição for verdadeira, o ciclo continua; se for falsa, o ciclo termina e a execução continua após o loop.

Este processo se repete até que a condição especificada não seja mais verdadeira, momento em que o loop **for** é encerrado e a execução continua com o código após o loop.

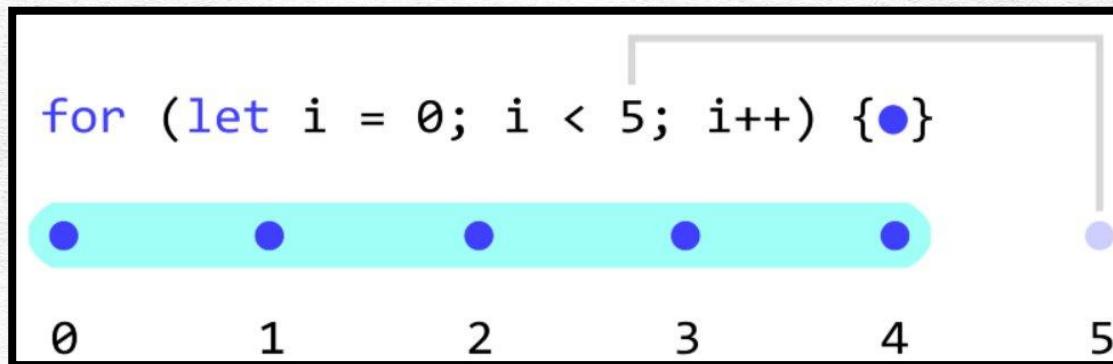


Aqui está um exemplo simples de um loop **for** em JavaScript:

```
for (let i = 0; i < 5; i++) {  
    console.log(i);  
}
```

Este loop imprimirá os números de 0 a 4 no console.

- A inicialização **let i = 0** define a variável **i** e a inicializa com o valor 0.
- A condição **i < 5** verifica se **i** é menor que 5. Enquanto isso for verdadeiro, o loop continuará executando.
- O incremento **i++** incrementa o valor de **i** em 1 após cada iteração.



Vamos analisar alguns exemplos práticos para explorarmos o conceito do FOR:

```
let produtos = ["Camiseta", "Calça", "Boné", "Meias"];  
  
for (let index = 0; index <= 3; index++) {  
    console.log("Produtos " + (index + 1) + ":" + produtos[index]);  
}
```

Este código cria um loop **for** que percorre um array chamado **produtos**, que contém itens de vestuário. Vamos dividir isso em partes:

- **Inicialização : let index = 0;**: Inicializa uma variável chamada **index** com o valor 0. Essa variável é usada para rastrear a posição atual dentro do array **produtos**.
- **Condição: index <= 3;**: Define a condição para o loop. Enquanto o valor de **index** for menor ou igual a 3, o loop continuará sendo executado. Isso significa que o loop irá percorrer os índices de 0 a 3 do array **produtos**.
- **Bloco de código: console.log("Produtos " + (index + 1) + ":" + produtos[index]);**: Isso imprime cada produto do array **produtos** no console.
 - "**Produtos** ": É uma string que precede o nome do produto.
 - **(index + 1)**: Adiciona 1 ao valor de **index**. Como os arrays em JavaScript começam com índice 0, isso faz com que a contagem dos produtos comece em 1.
 - **":":**: Adiciona dois pontos para separar o número do produto do seu nome.
 - **produtos[index]**: Acessa o produto atual no array **produtos** usando o valor de **index**.
- **Incremento: index++**: Após cada execução do bloco de código, o valor de **index** é incrementado em 1. Isso faz com que o loop avance para o próximo produto no array **produtos**.

No final, este loop **for** irá percorrer cada produto no array **produtos** e imprimir o nome do produto junto com um número correspondente, começando em 1. Isso ajuda a listar e numerar os produtos para facilitar a visualização.



Vamos analisar o próximo exemplo prático:

```
let vendas = [100, 150, 80, 200];
let totalVendas = 0;

for (let i = 0; i < vendas.length; i++) {
    totalVendas += vendas[i];
}

console.log("Total de vendas: " + totalVendas);
```

- **Inicialização:**
 - **let vendas = [100, 150, 80, 200];**: Cria um array chamado **vendas** que armazena os valores das vendas.
 - **let totalVendas = 0;**: Inicializa uma variável chamada **totalVendas** com o valor 0. Esta variável será usada para calcular o total das vendas.
- **Condição:**
 - **i < vendas.length;**: Define a condição para o loop. Enquanto o valor de **i** for menor que o tamanho do array **vendas**, o loop continuará sendo executado. Isso garante que o loop percorra todos os elementos do array.
- **Bloco de código:**
 - **totalVendas += vendas[i];**: Adiciona o valor do elemento atual do array **vendas** ao **totalVendas**. Isso ocorre em cada iteração do loop, o que resulta no cálculo do total das vendas.
- **Incremento:**
 - **i++**: Após cada execução do bloco de código, o valor de **i** é incrementado em 1. Isso faz com que o loop avance para o próximo elemento no array **vendas**.
- **Impressão no console:**
 - **console.log("Total de vendas: " + totalVendas);**: Após o loop ser concluído, o valor total das vendas é impresso no console. Então, no final, este loop **for** percorre cada elemento do array **vendas**, soma-os e imprime o total das vendas no console. Isso é útil para calcular o total de vendas em um determinado período de tempo.

Vamos analisar o último exemplo prático:

```
let vendas2 = [50, 200, 30, 400, 125];
let comissaoTotal = 0;

for (let venda = 0; venda < vendas2.length; venda++) {
    comissaoTotal += vendas2[venda] * 0.5;
}

console.log('Total de comissão: ${comissaoTotal}');
```

• Inicialização:

- **let vendas2 = [50, 200, 30, 400, 125];**: Cria um array chamado **vendas2** que armazena os valores das vendas.
- **let comissaoTotal = 0;**: Inicializa uma variável chamada **comissaoTotal** com o valor 0. Esta variável será usada para calcular o total da comissão.

• Condição:

- **venda < vendas2.length;**: Define a condição para o loop. Enquanto o valor de **venda** for menor que o tamanho do array **vendas2**, o loop continuará sendo executado. Isso garante que o loop percorra todos os elementos do array.

• Bloco de código:

- **comissaoTotal += vendas2[venda] * 0.5;**: Calcula a comissão para a venda atual e a adiciona ao **comissaoTotal**. A comissão é calculada multiplicando o valor da venda por 0.5 (ou seja, 50%). Isso é feito em cada iteração do loop.

• Incremento:

- **venda++**: Após cada execução do bloco de código, o valor de **venda** é incrementado em 1. Isso faz com que o loop avance para a próxima venda no array **vendas2**.

• Impressão no console:

- **console.log('Total de comissão: \${comissaoTotal}');**: Após o loop ser concluído, o valor total da comissão é impresso no console. Este loop **for** percorre cada venda no array **vendas2**, calcula a comissão para cada venda e soma-as para obter o total da comissão. O total da comissão é então impresso no console.

Um loop infinito em JavaScript (ou em qualquer linguagem de programação) ocorre quando uma estrutura de repetição, como um **for**, **while** ou **do-while**, continua executando indefinidamente, sem nunca atingir uma condição de término que permita a saída do loop. Isso pode acontecer por diversas razões, mas geralmente é devido a um erro na lógica do programa.

Loop infinitos podem ser problemáticos por algumas razões:

- **Consumo de recursos:** Um loop infinito pode consumir todos os recursos disponíveis do sistema, como memória ou poder de processamento da CPU, levando o sistema a travar ou ficar inutilizável.
- **Bloqueio de execução:** Como o loop nunca termina, qualquer código que venha depois dele no programa nunca será executado, o que pode causar a interrupção de outras funcionalidades do programa.
- **Impacto na performance:** Mesmo que o sistema não trave, um loop infinito pode prejudicar a performance do programa, já que recursos são desperdiçados em uma tarefa inútil.

Para evitar loops infinitos, é importante sempre garantir que as condições de saída dos loops sejam definidas corretamente e que haja um mecanismo para que o loop seja interrompido quando necessário. Isso pode envolver o uso de contadores, condições lógicas ou outros métodos para controlar o fluxo do programa dentro do loop. Além disso, é útil testar e revisar cuidadosamente o código para identificar e corrigir possíveis problemas de lógica que possam levar a loops infinitos.

```
JS loopinfinito.js
1  for (;;) {
2    |  console.log("Loop Infinito");
3  }
```

PROBLEMS OUTPUT DEBUG CONSOLE

Loop Infinito
Loop Infinito



Não utilizar os 3 argumentos do for:

Quando usamos o **for** em JavaScript, geralmente vemos algo assim:

```
for (let i = 0; i < 5; i++) {  
    console.log(i);  
}
```

Isso é muito comum e funciona bem. Mas podemos omitir alguns dos três argumentos. Por exemplo:

```
let i = 0;  
for (; i < 5;) {  
    console.log(i);  
    i++;  
}
```

- Aqui, nós movemos a inicialização de **i** para fora do próprio **for** e também removemos a parte do incremento. No entanto, a lógica do loop ainda é a mesma: enquanto **i** for menor que 5, ele vai continuar imprimindo **i** e incrementando-o.

Variável de inicialização no escopo global:

Normalmente, declaramos e inicializamos a variável de controle do loop dentro do próprio **for**, como em **for (let i = 0; ...**. Mas, tecnicamente, podemos declarar essa variável fora do **for**, colocando-a em um escopo mais amplo:

```
let i = 0;  
for (; i < 5; i++) {  
    console.log(i);  
}
```



No código anterior, **i** está sendo declarado fora do **for**. Isso significa que **i** estará disponível para qualquer código que venha antes ou depois do loop.

- **Não utilizar a parte do incremento:**

Da mesma forma que podemos omitir a inicialização, podemos omitir a parte do incremento:

```
for (let i = 0; i < 5;) {
    console.log(i);
    i++;
}
```

- Aqui, estamos fazendo o incremento manualmente dentro do bloco do loop, em vez de especificá-lo na parte do **for**. Assim como nos exemplos anteriores, isso ainda resulta no mesmo comportamento de um loop que incrementa **i** até que ele atinja 5.

Essas variações podem ser úteis em situações específicas, mas devem ser usadas com cuidado para garantir que o código permaneça legível e fácil de entender para outros programadores que possam trabalhar nele posteriormente.

Agora vamos conhecer uma estrutura de repetição chamada **WHILE**.

while (condição) {

Código que será executado
enquanto a condição for
verdadeira

}

WHILE

Essa estrutura define um bloco de código a ser executado repetidas vezes. Faz-se uma checagem da "condição de repetição" e, caso essa seja válida, uma iteração do bloco de código é executada.

Esse procedimento se repete até que a condição deixe de ser válida.

A diferença entre o **WHILE** e **Do...While**, é que no caso do **WHILE**, o bloco de instrução só será executado se a condição for verdadeira. Diferente do **Do...While**, que sempre terá pelo menos uma execução.



O **while** é outra estrutura de controle em JavaScript que permite executar um bloco de código repetidamente enquanto uma condição específica for verdadeira. Aqui está a sintaxe básica:

```
while (condição) {
    // Bloco de código a ser executado enquanto a condição for verdadeira
}
```

Aqui está como funciona:

- Primeiro, a condição é avaliada. Se a condição for verdadeira, o bloco de código dentro do **while** é executado. Se a condição for falsa desde o início, o bloco de código nunca será executado.
- Depois que o bloco de código dentro do **while** é executado, a condição é avaliada novamente. Se ainda for verdadeira, o bloco de código é executado novamente. Esse processo continua repetindo até que a condição se torne falsa.
- Quando a condição se torna falsa, a execução do código dentro do **while** é interrompida e o controle passa para a próxima linha de código após o bloco **while**.

Aqui está um exemplo simples de um **while** em JavaScript que imprime os números de 1 a 5:

```
let contador = 1;

while (contador <= 5) {
    console.log(contador);
    contador++;
}
```



1
2
3
4
5

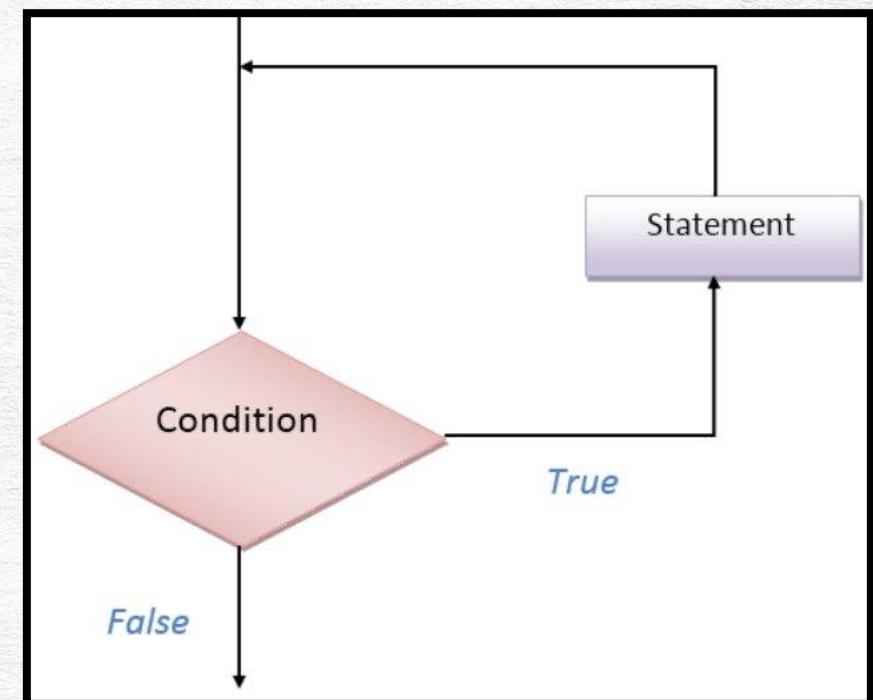


No exemplo do slide anterior, **contador** é inicializado como 1. Enquanto **contador** for menor ou igual a 5, o bloco de código dentro do **while** é executado, imprimindo o valor de **contador** e então incrementando-o. Assim que **contador** atinge 6, a condição se torna falsa e a execução do **while** é interrompida.

Vamos explicar o fluxo do **while**:

- **Avaliação da Condição Inicial:** Antes de entrar no bloco de código do **while**, a condição é avaliada. Se a condição inicialmente for falsa, o bloco de código dentro do **while** não será executado.
- **Entrando no Loop:** Se a condição inicial for verdadeira, o bloco de código dentro do **while** é executado.
- **Avaliação da Condição Após a Execução do Bloco de Código:** Após a execução do bloco de código, a condição é avaliada novamente. Se ainda for verdadeira, o bloco de código é executado novamente.
- **Repetição do Passo 3:** O processo de execução do bloco de código e avaliação da condição continua repetindo enquanto a condição permanecer verdadeira.
- **Fim do Loop:** Quando a condição se torna falsa, a execução do bloco de código dentro do **while** é interrompida e o controle passa para a próxima linha de código após o bloco **while**.

Então, em resumo, o **while** é uma estrutura de controle de fluxo que executa repetidamente um bloco de código enquanto uma condição especificada permanecer verdadeira. Ele é útil quando você precisa repetir uma determinada ação várias vezes, mas não sabe quantas vezes será necessário repeti-la antecipadamente.



O loop while é uma estrutura de repetição em JavaScript que executa um bloco de código enquanto uma condição especificada for avaliada como verdadeira. Ao contrário do do...while, o while verifica a condição antes de executar o bloco de código.

Estrutura básica do "while":

- **variável**: uma variável de controle usada para monitorar o estado do loop.
- **condição**: uma expressão avaliada antes de cada iteração; se for verdadeira, o loop continua.

```
let variavel = valorInicial;

while (condicao) {
    // Código a ser executado enquanto a condição for verdadeira
}
```

Exemplo 1: Contando de 1 a 5

Vamos criar um programa que exibe no console os números de 1 a 5 usando o loop while.

```
let numero = 1;

while (numero <= 5) {
    console.log("Número: " + numero);
    numero++; // numero + 1
}

console.log("Contagem com while concluída!");
```

Análise do exemplo:

- O programa inicializa a variável numero com o valor 1.
- Antes de cada execução do bloco de código, a condição numero <= 5 é verificada:
 - Se for verdadeira, o bloco é executado.
 - Se for falsa, o loop termina.
- Dentro do loop, o valor de numero é exibido no console e, em seguida, incrementado em 1.
- Quando numero atinge o valor 6, a condição se torna falsa e o programa segue para o comando após o loop.



Exemplo 2: Registrando Funcionários

Imagine que você precisa registrar as horas de 3 funcionários. Cada registro é exibido no console e, ao final, é informada a conclusão do processo.

```
let funcionariosRegistrados = 0;

while (funcionariosRegistrados < 3) {
    console.log(
        "Funcionário " + (funcionariosRegistrados + 1) + " registrou suas horas."
    );
    funcionariosRegistrados++;
}

console.log("Fim de registro");
```

Análise do exemplo:

- O programa inicia com a variável `funcionariosRegistrados` definida como 0.
- A condição `funcionariosRegistrados < 3` é avaliada antes de cada iteração:
 - Se for verdadeira, o bloco de código é executado.
 - Se for falsa, o loop termina.
- A cada iteração, uma mensagem é exibida no console e `funcionariosRegistrados` é incrementado.
- Quando `funcionariosRegistrados` atinge 3, a condição se torna falsa e o loop é finalizado.

Quando usar o "while"?

- Quando você quer repetir um bloco de código enquanto uma condição específica for verdadeira.
- Para situações em que não é necessário executar o código pelo menos uma vez (diferente do `do...while`).



Embora **for** e **while** sejam estruturas de controle de fluxo diferentes em JavaScript, eles têm algumas semelhanças em termos de funcionalidade e finalidade:

- **Repetição Controlada por Condição:** Ambos **for** e **while** permitem que você execute um bloco de código repetidamente com base em uma condição específica. A diferença principal é que o **for** geralmente é usado quando você sabe o número exato de iterações que deseja fazer, enquanto o **while** é mais comumente usado quando o número de iterações não é conhecido antecipadamente.
- **Avaliação de Condição:** Ambos os loops avaliam uma condição antes de executar o bloco de código associado a eles. Se a condição for verdadeira, o bloco de código é executado; caso contrário, o bloco de código é ignorado e o programa continua sua execução.
- **Possibilidade de Laços Infinitos:** Tanto **for** quanto **while** têm o potencial de criar loops infinitos se a condição nunca se tornar falsa. Isso geralmente é indesejado e pode causar travamento ou congelamento do programa. Portanto, é importante garantir que a condição de parada seja alcançada em algum momento.
- **Flexibilidade:** Embora o **for** seja geralmente usado quando se sabe o número exato de iterações, ele ainda oferece flexibilidade para expressar a condição de controle e a ação de atualização em uma única linha. Da mesma forma, o **while** oferece flexibilidade ao permitir que a condição seja qualquer expressão JavaScript válida que resulte em um valor booleano.

Agora vamos analisar alguns exemplos aplicando a estrutura while:

```
let produtos = ["Camiseta", "Calça", "Boné", "Meias"]; // 0, 1, 2, 3
let index = 0;

while (index < produtos.length) {
  console.log(`Produto ${index + 1} : ${produtos[index]}`);
  index++;
}
```

- **Inicialização das Variáveis:**
 - **produtos** é uma matriz (array) que contém quatro elementos: "Camiseta", "Calça", "Boné" e "Meias".
 - **index** é uma variável inicializada com o valor 0. Ela será usada para percorrer a matriz **produtos**.
- **Condição do Loop:** A condição do **while** é **index < produtos.length**, o que significa que o bloco de código dentro do **while** será executado enquanto o valor de **index** for menor que o comprimento da matriz **produtos**.
- **Execução do Bloco de Código:** Dentro do bloco de código do **while**, temos:
 - **console.log(Produto \${index + 1} : \${produtos[index]})**: Isso imprime o nome do produto atual na posição **index** da matriz **produtos**. A expressão **index + 1** é usada para mostrar o número do produto começando em 1 ao invés de 0.
 - **index++**: Isso incrementa o valor de **index** em 1 após a impressão do produto atual, movendo para o próximo produto na próxima iteração.
- **Atualização da Condição:** Após a execução do bloco de código dentro do **while**, o valor de **index** é incrementado em 1.
- **Repetição:** O bloco de código dentro do **while** será executado repetidamente enquanto a condição **index < produtos.length** for verdadeira, ou seja, enquanto ainda houver produtos não impressos na matriz **produtos**.
- **Fim do Loop:** Quando **index** se torna igual ao comprimento da matriz **produtos**, a condição **index < produtos.length** se torna falsa e o loop termina.

O próximo exemplo aplicando a estrutura while:

```
let vendas = [100, 150, 80, 200]; //530
let totalVendas = 0;
let i = 0;

while (i < vendas.length) {
    // 4
    totalVendas += vendas[i]; // totalVendas = totalVendas + vendas[index];
    i++;
}
console.log("Total de vendas: " + totalVendas);
```

• **Inicialização das Variáveis:**

• **vendas**: É uma matriz (array) que contém os valores de vendas para cada produto. No exemplo, são [100, 150, 80, 200].

• **totalVendas**: É uma variável inicializada com o valor 0. Ela será usada para calcular o total de vendas.

• **i**: É uma variável de controle inicializada com o valor 0. Ela será usada como índice para percorrer a matriz **vendas**.

• **Condição do Loop**: A condição do **while** é **i < vendas.length**, o que significa que o bloco de código dentro do **while** será executado enquanto o valor de **i** for menor que o comprimento da matriz **vendas**.

• **Execução do Bloco de Código**: Dentro do bloco de código do **while**, temos:

• **totalVendas += vendas[i]**: Isso adiciona o valor de vendas do produto na posição **i** da matriz **vendas** à variável **totalVendas**.

Ou seja, estamos acumulando o valor de todas as vendas.

• **i++**: Isso incrementa o valor de **i** em 1 após a adição do valor da venda atual, movendo para a próxima posição na próxima iteração.

• **Atualização da Condição**: Após a execução do bloco de código dentro do **while**, o valor de **i** é incrementado em 1.

• **Repetição**: O bloco de código dentro do **while** será executado repetidamente enquanto a condição **i < vendas.length** for verdadeira, ou seja, enquanto ainda houver vendas não somadas na matriz **vendas**.

• **Fim do Loop**: Quando **i** se torna igual ao comprimento da matriz **vendas**, a condição **i < vendas.length** se torna falsa e o loop termina.

• **Impressão do Total de Vendas**: Após o término do loop, o código imprime o total de vendas acumulado na variável **totalVendas**.

O último exemplo aplicando a estrutura while para analisarmos:

```
let vendas2 = [50, 200, 30, 400, 125]; //805
// 0.5%
let comissaoTotal = 0;
index = 0;

while (index < vendas2.length) {
    comissaoTotal += vendas2[index] * 0.5;
    index++;
}
console.log("Valor da comissão total: " + comissaoTotal);
```

• Inicialização das Variáveis:

- **vendas2**: É uma matriz (array) que contém os valores das vendas para cada produto. No exemplo, são [50, 200, 30, 400, 125].
- **comissaoTotal**: É uma variável inicializada com o valor 0. Ela será usada para calcular o valor total da comissão.
- **index**: É uma variável de controle inicializada com o valor 0. Ela será usada como índice para percorrer a matriz **vendas2**.

• Condição do Loop:

A condição do **while** é **index < vendas2.length**, o que significa que o bloco de código dentro do **while** será executado enquanto o valor de **index** for menor que o comprimento da matriz **vendas2**.

• Execução do Bloco de Código:

Dentro do bloco de código do **while**, temos:

- **comissaoTotal += vendas2[index] * 0.5**: Isso calcula a comissão para a venda atual, multiplicando o valor da venda na posição **index** da matriz **vendas2** por 0.5 (50%). A comissão é então adicionada ao **comissaoTotal**.
- **index++**: Isso incrementa o valor de **index** em 1 após o cálculo da comissão atual, movendo para a próxima posição na próxima iteração.

• Atualização da Condição:

Após a execução do bloco de código dentro do **while**, o valor de **index** é incrementado em 1.

• Repetição:

O bloco de código dentro do **while** será executado repetidamente enquanto a condição **index < vendas2.length** for verdadeira, ou seja, enquanto ainda houver vendas não computadas na matriz **vendas2**.

• Fim do Loop:

Quando **index** se torna igual ao comprimento da matriz **vendas2**, a condição **index < vendas2.length** se torna falsa e o loop termina.

• Impressão do Valor Total da Comissão:

Após o término do loop, o código imprime o valor total da comissão acumulado na variável **comissaoTotal**.

DO ... WHILE

A estrutura **Do... While** vem da expressão em inglês, **FAÇA ...ENQUANTO.**

Estrutura que define um bloco de código a ser executado repetidas vezes. Ao término de cada **iteração (cada repetição)**, a **"condição de repetição"** é checada, e caso seja válida, mais uma **iteração** executada. A condição é verificada após os comandos do bloco serem executados, ou seja, mesmo que a condição seja falsa, é garantia que o bloco será executado ao menos uma vez

do {

Código que será executado
enquanto a condição for
verdadeira

} while (condição)

Já que o **Do...While** executa o bloco e depois confere a validez da condição de repetição, então isso significa que o bloco será executado **PELO MENOS 1 VEZ.**

A condição que utilizamos é uma **condição booleana**, que retornará true ou false para o código saber se ele realizará a repetição ou não.



O do-while é uma estrutura de controle de fluxo semelhante ao while, mas com uma diferença fundamental: no do-while, o bloco de código é executado pelo menos uma vez, mesmo que a condição especificada seja falsa desde o início. Aqui está uma explicação passo a passo sobre o do-while em JavaScript:

```
do {  
    // Bloco de código a ser executado  
} while (condição);
```

- **Execução Inicial:**
 - O bloco de código dentro do **do** é executado uma vez antes que a condição seja verificada. Isso significa que, mesmo que a condição seja falsa desde o início, o bloco de código dentro do **do** será executado pelo menos uma vez.
 - **Avaliação da Condição:**
 - Após a execução do bloco de código dentro do **do**, a condição especificada no **while** é avaliada. Se a condição for verdadeira, o bloco de código dentro do **do** será executado novamente. Se a condição for falsa, o loop termina e a execução continua após o **do-while**.
 - **Repetição:**
 - O bloco de código dentro do **do** será repetidamente executado enquanto a condição especificada no **while** for verdadeira.
 - **Fim do Loop:**
 - Quando a condição se torna falsa, a execução do bloco de código dentro do **do-while** é interrompida e o controle passa para a próxima linha de código após o **do-while**.
- O **do-while** é útil quando você precisa garantir que um bloco de código seja executado pelo menos uma vez, independentemente da condição. Ele é comumente usado em situações em que a execução de um bloco de código é necessária antes de verificar a condição de término.

Vamos explorar o fluxo do **do-while**:

- **Execução Inicial:**

- O bloco de código dentro do **do** é executado antes de verificar a condição especificada no **while**. Isso significa que o bloco de código será executado pelo menos uma vez, independentemente da condição ser verdadeira ou falsa inicialmente.

- **Avaliação da Condição:**

- Após a execução do bloco de código dentro do **do**, a condição especificada no **while** é avaliada. Se a condição for verdadeira, o bloco de código dentro do **do** será executado novamente. Se a condição for falsa, o loop termina.

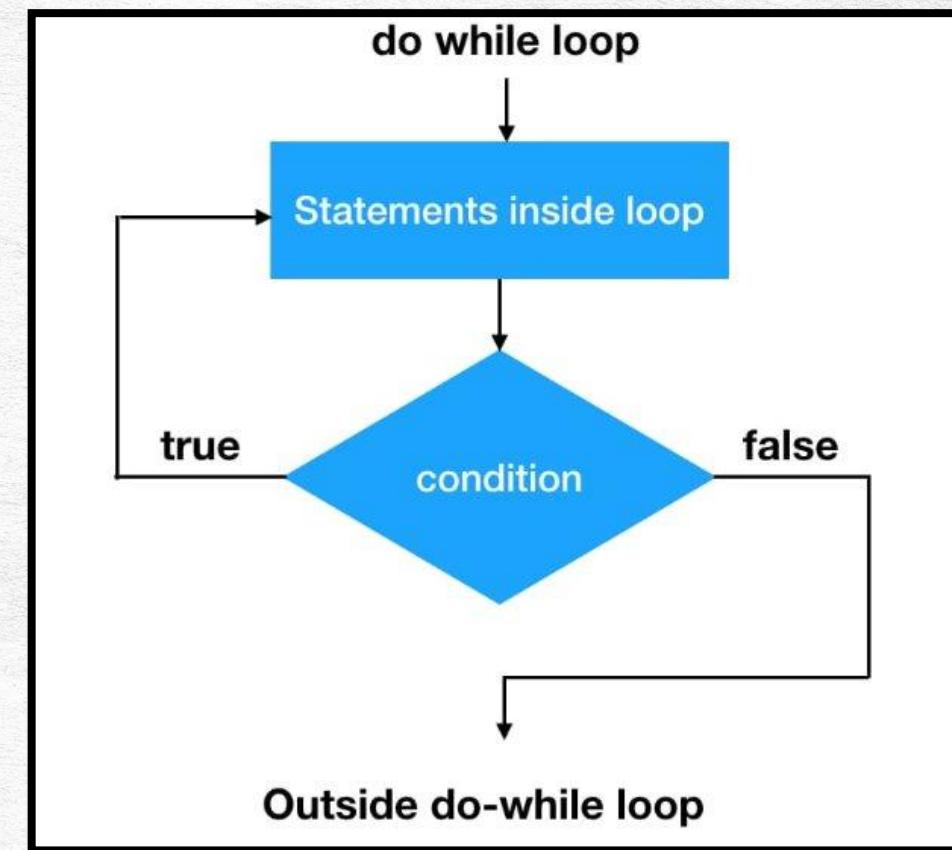
- **Repetição:**

- O bloco de código dentro do **do** continuará a ser executado repetidamente enquanto a condição especificada no **while** for verdadeira.

- **Fim do Loop:**

- Quando a condição se torna falsa, a execução do bloco de código dentro do **do-while** é interrompida e o controle passa para a próxima linha de código após o **do-while**.

Em resumo, o **do-while** garante que o bloco de código dentro do **do** seja executado pelo menos uma vez, independentemente da condição. Depois, ele continua a executar o bloco de código enquanto a condição for verdadeira. É uma estrutura útil quando você precisa garantir uma execução inicial antes de verificar a condição de término.



Vamos comparar as diferenças entre **while** e **do-while**:

- **Momento da Avaliação da Condição:**

- No **while**, a condição é avaliada antes da execução do bloco de código. Se a condição for falsa desde o início, o bloco de código nunca será executado.
- No **do-while**, o bloco de código é executado pelo menos uma vez, independentemente da condição ser verdadeira ou falsa inicialmente. A condição é avaliada após a execução do bloco de código.

- **Uso Prevalente:**

- O **while** é frequentemente usado quando a execução do bloco de código depende da condição ser verdadeira desde o início. Se a condição inicialmente for falsa, o bloco de código nunca será executado.
- O **do-while** é usado quando você precisa garantir que o bloco de código seja executado pelo menos uma vez, independentemente da condição. É útil quando você quer que um bloco de código seja executado antes de verificar a condição de término.

- **Comportamento em Laços Infinitos:**

- Se a condição de um **while** nunca se tornar falsa, o loop se tornará infinito, pois o bloco de código nunca será executado.
- No **do-while**, como o bloco de código é executado antes da avaliação da condição, mesmo que a condição seja falsa inicialmente, o bloco de código será executado pelo menos uma vez, evitando um verdadeiro laço infinito.

O loop do...while é uma estrutura de repetição em JavaScript que garante que o bloco de código seja executado pelo menos uma vez, independentemente da condição ser verdadeira ou falsa no início. A verificação da condição ocorre apenas ao final de cada iteração.

Estrutura básica do "do...while":

```
let variavel = valorInicial;  
  
do {  
    // Código a ser executado  
} while (condicao);
```

variavel: uma variável de controle que pode ser atualizada a cada iteração.

condicao: uma expressão avaliada ao final do loop; se for verdadeira, o loop continua.

Exemplo 1: Contando de 1 a 5

Vamos criar um programa que exibe no console os números de 1 a 5 usando o loop do...while.

Análise do exemplo:

- O programa inicializa a variável numero com o valor 1.
- O bloco de código é executado pelo menos uma vez, exibindo o valor atual de numero.
- Em seguida, numero é incrementado (numero++).
- A condição numero <= 5 é avaliada:
 - Se for verdadeira, o loop continua.
 - Se for falsa, o loop termina e o programa segue para o próximo comando após o loop.

```
let numero = 1;  
  
do {  
    console.log("Número" + numero);  
    numero++;  
} while (numero <= 5);  
  
console.log("Contagem do.. while foi concluída!");
```



Exemplo 2: Registrando Funcionários

Imagine que você precisa registrar a entrada de um grupo de 6 funcionários. Cada registro é exibido no console e, ao final, é informada a conclusão do processo.

Análise do exemplo:

- O programa inicia com a variável `funcionariosParaRegistrar` definida como 6.
- O loop exibe uma mensagem para cada funcionário, decrementando o valor da variável em cada iteração.
- O processo continua enquanto `funcionariosParaRegistrar` for maior que 0.
- Quando a condição se torna falsa, o loop termina e uma mensagem final é exibida no console.

```
let funcionariosParaRegistrar = 6; // total

do {
    console.log(
        "Registrando a entrada do funcionário " + funcionariosParaRegistrar
    );
    funcionariosParaRegistrar--; // -1
} while (funcionariosParaRegistrar > 0);

console.log("Nenhum funcionário restante para registrar!");
```

Quando usar o "do...while"?

- Quando você precisa garantir que o bloco de código seja executado pelo menos uma vez.
- Para cenários onde a verificação da condição depende de uma execução inicial do código (por exemplo, interações baseadas em entrada do usuário).

Compreender o funcionamento do `do...while` e suas aplicações é um passo importante para dominar as estruturas de controle em JavaScript. Experimente os exemplos acima e tente adaptá-los a diferentes cenários!



Vamos comparar as diferenças entre **while** e **do-while**, através de um exemplo prático:

```
let numero = 5;

do {
  console.log("Executa mesmo que a condição seja falsa");
} while (numero === 6);

while (numero === 6) {
  console.log("Executa");
}
```

```
PS C:\Users\mikag\OneDrive\Documentos\AulasJavascript\EstruturasRepeticao\Aula 15> node ./exemplo.js
Executa mesmo que a condição seja falsa
```

Vamos analisar a estrutura **do-while** com um exemplo prático:

```
PS C:\Users\mikag\OneDrive\Documentos\AulasJavascript\EstruturasRepeticao\Aula 15> node ./doWhile.js
Produto 1: Camiseta
Produto 2: Calça
Produto 3: Boné
Produto 4: Meias
```

```
let produtos = ["Camiseta", "Calça", "Boné", "Meias"] // 0, 1, 2, 3

let index = 0;

do {
    console.log("Produto " + (index + 1) + ": " + produtos[index]);
    index++;
} while (index < produtos.length);
```

Inicialização das variáveis:

- **produtos** é uma array que contém uma lista de produtos.
- **index** é uma variável que inicialmente é definida como 0. Essa variável será usada para acessar cada elemento da array **produtos**.

Início do laço do...while:

- **do {** indica o início do bloco de código que será executado pelo laço.
- Aqui, primeiro o código dentro do bloco é executado, e depois é feita a verificação da condição.

Corpo do laço:

- **console.log("Produto " + (index + 1)+": " + (produtos[index]));**: Isso imprime na tela o produto atual, junto com o seu índice na array **produtos**. Note que **index + 1** é utilizado para exibir o número do produto de forma mais amigável para os usuários, já que os índices em JavaScript começam de 0.
- **index++;**: Incrementa o valor da variável **index** em 1. Isso é feito para que, na próxima iteração do laço, o próximo produto da array **produtos** seja acessado.

Condição de saída do laço:

- **while (index < produtos.length);**: Esta é a condição de saída do laço. Enquanto o valor de **index** for menor do que o comprimento (número de elementos) da array **produtos**, o laço continuará executando.

Vamos observar outro exemplo prático da estrutura **do-while**:

Inicialização das variáveis:

- **vendas**: é uma array que contém os valores das vendas.
- **totalVendas**: é uma variável que armazenará a soma total das vendas.
- **indexVendas**: é uma variável que começa em 0 e será usada para percorrer a array **vendas**.

Início do laço do...while:

- **do {** indica o início do bloco de código que será repetido pelo laço.
- Dentro desse bloco, primeiro a soma das vendas é atualizada na variável **totalVendas** usando **totalVendas += vendas[indexVendas]**; Isso adiciona o valor da venda atual ao total.
- Em seguida, **indexVendas++**; incrementa o valor da variável **indexVendas** para que possamos avançar para o próximo elemento na próxima iteração.

Condição de continuação do laço:

- **while (indexVendas < vendas.length)**: Esta é a condição que verifica se ainda há elementos na array de vendas para processar. Enquanto o valor de **indexVendas** for menor que o comprimento da array **vendas**, o laço continuará executando.

Fim do laço:

- **}**: Indica o fim do bloco de código do laço.

Impressão do total de vendas:

- **console.log ("Total de vendas: " + totalVendas)**: Fora do laço, o total das vendas é impresso no console.

```
let vendas = [100, 150, 80, 200]; //530
let totalVendas = 0;
let indexVendas = 0;

do {
    totalVendas += vendas[indexVendas];
    indexVendas++;
} while (indexVendas < vendas.length);

console.log("Valor total de vendas " + totalVendas);
```

Vamos para o último exemplo prático da estrutura **do-while**:

Inicialização das variáveis:

- **vendas2**: é uma array que contém os valores das vendas.
- **comissaoTotal**: é uma variável que armazenará o total da comissão.
- **indexVendas**: é uma variável que começa em 0 e será usada para percorrer a array **vendas2**.

Laço do...while:

- **do {** indica o início do bloco de código que será repetido pelo laço.

Cálculo da comissão:

- **comissaoTotal += vendas2[indexVendas] * 0.5;**: Aqui, para cada venda, calculamos a comissão multiplicando o valor da venda pelo percentual de comissão (0.5, que representa 50% do valor da venda). Esse valor é então adicionado ao total de comissões.

Incremento do índice:

- **indexVendas++;**: A cada iteração do loop, incrementamos o valor de **indexVendas** para que possamos acessar o próximo elemento da array **vendas2**.

Condição de continuação do laço:

- **while (indexVendas < vendas2.length)**: Esta é a condição que verifica se ainda há elementos na array de vendas para processar. Enquanto o valor de **indexVendas** for menor que o comprimento da array **vendas2**, o laço continuará executando.

Impressão do total da comissão:

- **console.log ("Valor da comissão total: " + comissaoTotal)**: Após o loop, o valor total da comissão é impresso no console.

```
let vendas2 = [50, 200, 30, 400, 125]; //805
// 0.5%

let comissaoTotal = 0;
indexVendas = 0;

do {
    comissaoTotal += vendas2[indexVendas] * 0.5;
    indexVendas++;
} while (indexVendas < vendas2.length);

console.log("Comissão total de vendas :" + comissaoTotal);
```

Na programação existem muitas formas diferentes de você resolver um problema., assim como utilizamos estruturas diferentes para criar o exercício anterior de Fibonacci. Desta vez, iremos utilizar uma outra técnica chamada Recursividade.

RECURSIVIDADE

Recursividade é um técnica de programação que consiste em ter estrutura (uma função, por exemplo) invocando a saí mesma. Uma função, por exemplo, pode solicitar outra invocação dela mesma para resolver um problema.



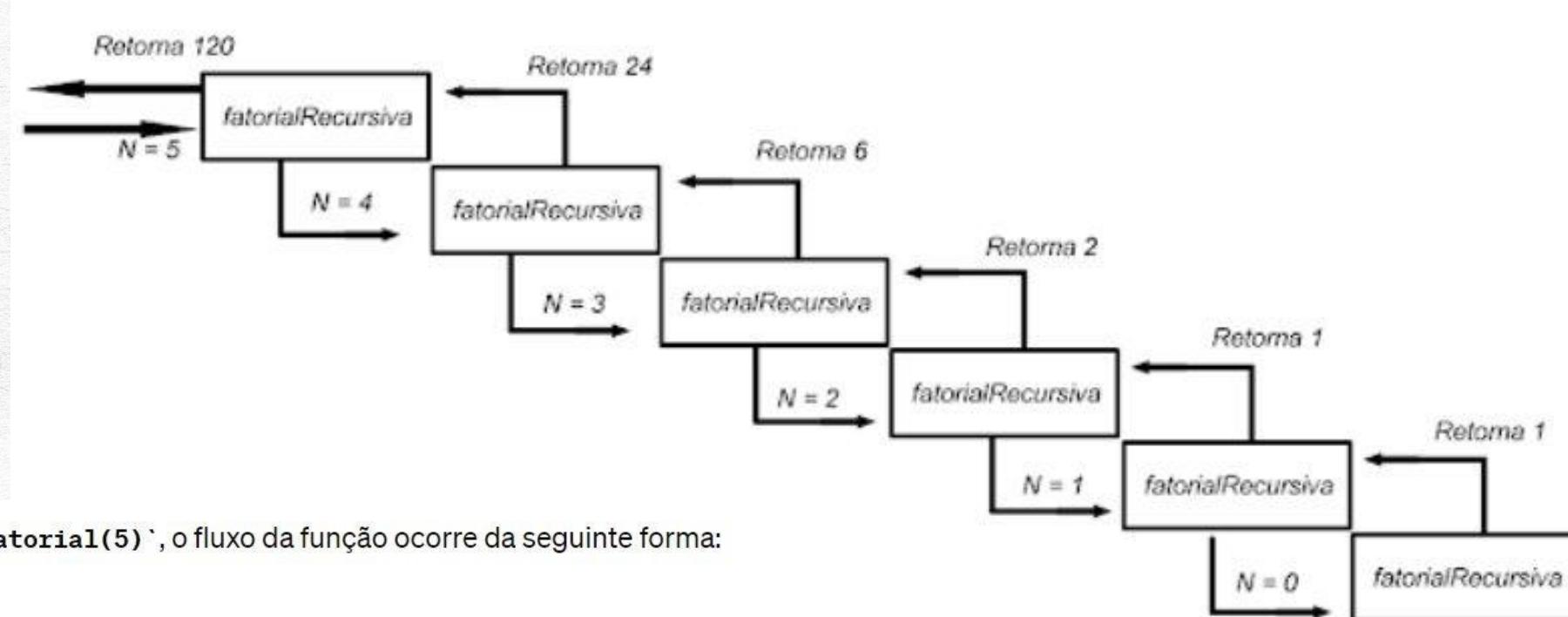
```
function calcularTemperaturaDaRuaHoje(dataDeHoje){  
    ...  
    calcularTemperaturaDaRuaHoje(dataDeOntem);  
}
```

Um exemplo clássico de função recursiva é o cálculo do fatorial de um número.

```
0! = 1
1! = 1
2! = 2 * 1
3! = 3 * 2 * 1
4! = 4 * 3 * 2 * 1
5! = 5 * 4 * 3 * 2 * 1
6! = 6 * 5 * 4 * 3 * 2 * 1
7! = 7 * 6 * 5 * 4 * 3 * 2 * 1
8! = 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1
9! = 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1
10! = 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1
```



```
5 (5 * 4 * 3 * 2 * 1)
```



No exemplo acima, quando chamamos `fatorial(5)`, o fluxo da função ocorre da seguinte forma:

- `fatorial(5)` chama `fatorial(4)`.
- `fatorial(4)` chama `fatorial(3)`.
- `fatorial(3)` chama `fatorial(2)`.
- `fatorial(2)` chama `fatorial(1)`.
- `fatorial(1)` retorna 1 (caso base).
- A partir daí, cada chamada recursiva retorna um valor e a multiplicação ocorre à medida que as chamadas recursivas desenrolam.

```
function fatorial(n) {  
    // Caso Base: Se n for 0 ou 1, retornar 1  
    if (n === 0 || n === 1) {  
        return 1;  
    }  
    // Chamada Recursiva: Se n for maior que 1, chamar a função fatorial novamente  
    else {  
        return n * fatorial(n - 1);  
    }  
}
```

- **Caso base:** Se o número fornecido for 0 ou 1, então o factorial é 1. Essa é a condição de parada da recursão.
- **Caso recursivo:** Se o número fornecido for maior que 1, a função chama a si mesma com o argumento **n - 1**. Ou seja, ela se chama recursivamente até que o argumento seja reduzido ao caso base.

```
console.log(fatorial(5)); // Saída: 120 (5 * 4 * 3 * 2 * 1)
```

Quando esse código é executado, a função **fatorial(5)** é chamada:

- **fatorial(5)** chama **fatorial(4)**
- **fatorial(4)** chama **fatorial(3)**
- **fatorial(3)** chama **fatorial(2)**
- **fatorial(2)** chama **fatorial(1)**
- **fatorial(1)** retorna 1 (caso base)
- **fatorial(2)** retorna $2 * 1 = 2$
- **fatorial(3)** retorna $3 * 2 = 6$
- **fatorial(4)** retorna $4 * 6 = 24$
- **fatorial(5)** retorna $5 * 24 = 120$



O JavaScript é *single thread*, isso significa que ele pode lidar apenas com uma tarefa por vez, em outras palavras, um pedaço de código por vez. Ele possui uma única **Call Stack** (Pilha de execução) e juntamente com outras partes como **Head** e **Queue** (Fila) formam o JavaScript Concurrency Model.

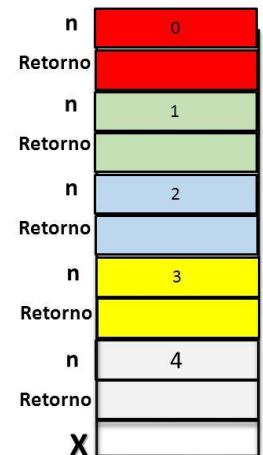
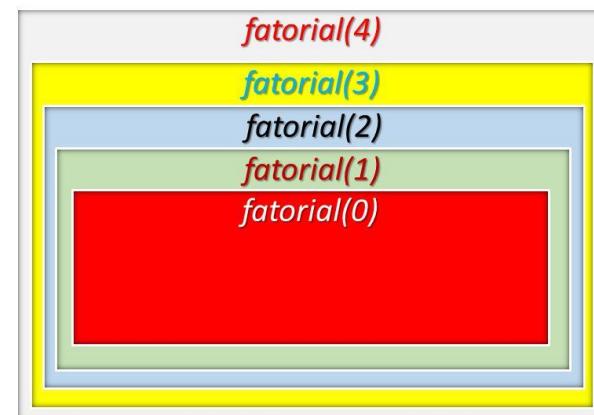
Call Stack

É uma estrutura de dados que registra as chamadas de funções, basicamente se chamamos uma função, então ela é adicionada na pilha e quando nos retornamos de uma função, a mesma é retirada da pilha.

Para cada nova ação que precisa ser executada, esta é “empilhada” e terá que ser processada antes da ação anterior voltar a ser trabalhada.

Note que as pilhas não tem tamanho infinito e quando você cria uma função recursiva muito grande ou mau feita, causará o famoso erro que chamamos de stack-overflow.

Recursão e a Pilha de Execução (*stack*)



Quais as vantagens da Recursividade?

Essa é uma dúvida bem comum, devo iterar? ou usar a recursão? Nos seguintes casos, seria melhor usar a recursão:

- Reduzir o código e torna-lo mais simples de ler:

Em grande maioria dos casos é possível escrever a mesma lógica com um número menor de linhas, tornando-o mais fácil de entender e debugar.

- Quando você está trabalhando com uma linguagem funcional:

Linguagens funcionais lidam com recursão muito melhor que imperativas. Já as linguagens imperativas lidam com iteração melhor.

- A recursão é melhor na travessia de árvores. Uma versão extremamente simplificada do que isso significa é a seguinte:

Uma árvore é uma coleção de objetos que estão ligados um ao outro, e através da recursividade conseguimos caminhar pelas folhas e ramificações para encontrar o que precisamos.

Árvore de chamadas

Uma **árvore de chamadas** em JavaScript é uma representação visual das chamadas de função que ocorrem durante a execução de um programa. Ela mostra a **hierarquia das chamadas de função**, onde cada nó da árvore representa uma chamada de função e suas respectivas informações.

Quando uma função é chamada em JavaScript, um novo contexto de execução é criado e adicionado à pilha de chamadas (também conhecida como pilha de execução). Esse contexto de execução contém informações sobre a função em execução, como os parâmetros passados e as variáveis locais.

A **árvore de chamadas** começa com a função principal (a função que é chamada inicialmente) como a raiz da árvore. À medida que o programa é executado e novas funções são chamadas, novos nós são adicionados à árvore, representando as chamadas de função subsequentes.

Cada **nó da árvore** de chamadas possui uma relação pai-filho, onde o nó pai representa a função que chamou a função atual e o nó filho representa a função que está sendo chamada. Isso cria uma estrutura hierárquica, onde as chamadas de função podem ser rastreadas de volta à função principal.

A **árvore de chamadas** é útil para entender o fluxo de execução do programa e visualizar as chamadas de função em diferentes níveis de aninhamento. Ela também pode ser usada para identificar problemas de recursividade ou estouro de pilha, pois mostra a quantidade de chamadas de função que estão sendo empilhadas na pilha de execução.

A recursividade em JavaScript pode ocasionar alguns problemas se não for utilizada corretamente. Aqui estão alguns dos problemas mais comuns:

- **Estouro de pilha (Stack Overflow):** Se uma função recursiva não tiver uma condição de parada adequada ou se o número de chamadas recursivas for muito grande, pode ocorrer um estouro de pilha. Isso acontece quando a pilha de chamadas recursivas fica cheia e não há mais espaço para adicionar novas chamadas. Isso pode levar ao encerramento abrupto do programa.
- **Desempenho lento:** Em alguns casos, a recursividade pode ser menos eficiente em termos de desempenho em comparação com uma solução iterativa. Isso ocorre porque cada chamada recursiva envolve a criação de um novo contexto de execução e empilhamento de chamadas na pilha de execução. Isso pode resultar em um tempo de execução mais longo e maior consumo de recursos.
- **Dificuldade de depuração:** A recursividade pode tornar a depuração do código mais complexa. Como cada chamada recursiva cria um novo contexto de execução, pode ser difícil acompanhar o fluxo de execução e identificar erros ou problemas de lógica.
- **Uso excessivo de memória:** Dependendo da implementação, a recursividade pode consumir uma quantidade significativa de memória. Cada chamada recursiva adiciona um novo contexto de execução à pilha, o que pode levar a um uso excessivo de memória, especialmente para problemas com muitas chamadas recursivas ou profundidade recursiva.

Portanto, ao utilizar a recursividade em JavaScript, é importante ter cuidado para evitar esses problemas. É necessário garantir que a função recursiva tenha uma condição de parada adequada, limitar o número de chamadas recursivas e considerar a eficiência em termos de desempenho e uso de memória.

O que é Recursividade?

Recursividade é uma técnica de programação onde uma função chama a si mesma para resolver um problema. Essa abordagem é frequentemente usada quando um problema pode ser dividido em subproblemas menores e semelhantes ao original.

Uma função recursiva geralmente possui:

- **Caso base:** uma condição que encerra as chamadas recursivas.
- **Chamada recursiva:** a função se chama novamente com uma versão reduzida ou modificada do problema.

Estrutura básica de uma função recursiva:

```
function funcaoRecursiva(parametro) {  
    if (condicaoDeParada) {  
        return resultado;  
    }  
    return resultadoParcial + funcaoRecursiva(parametroModificado);  
}
```



Exemplo Prático: Contando Caracteres de uma String

Vamos criar uma função que calcula a quantidade de caracteres em uma string utilizando recursividade.

```
function recursividade(string) {  
    if (string === "") { // Caso base: string vazia  
        return 0;  
    }  
    return 1 + recursividade(string.substring(1)); // Chamada recursiva com a string  
}  
  
console.log(recursividade("Hashtag")); // 7  
console.log(recursividade("H")); // 1  
console.log(recursividade("")) // 0
```

Saída no console:

7
1
0

Análise do exemplo:

- **Entrada inicial:**
 - A função é chamada com a string "Hashtag".
- **Processo recursivo:**
 - A cada chamada, a string é reduzida removendo o primeiro caractere (com `substring(1)`).
 - Uma unidade é adicionada ao resultado parcial (representando o caractere removido).
- **Caso base:**
 - Quando a string se torna vazia (""), a função retorna 0, encerrando a recursão.
- **Resultado final:**
 - As chamadas retornam seus valores acumulados até a função inicial.

Visualização da Árvore de Chamadas:

```
recursividade("Hashtag")
|
+-- recursividade("ashtag")
  |
  +-- recursividade("shtag")
    |
    +-- recursividade("htag")
      |
      +-- recursividade("tag")
        |
        +-- recursividade("ag")
          |
          +-- recursividade("g")
            |
            +-- recursividade("") - Parada
```

Como a recursão funciona?

- Cada chamada cria um novo "quadro" na memória, que guarda o estado atual da função.
- Quando o caso base é alcançado, os resultados começam a ser retornados de forma regressiva.

Exemplo com Entrada Vazia:

Caso a entrada seja uma string vazia (""), a condição de parada é imediatamente atendida e a função retorna 0.

Quando usar Recursividade?

- Quando o problema pode ser dividido em subproblemas semelhantes.
- Em estruturas de dados hierárquicas, como árvores e grafos.
- Para problemas que seguem padrões repetitivos, como fatorial, sequência de Fibonacci e busca em profundidade.

Trabalhando com Objetos e Recursividade

Recursividade também pode ser usada para trabalhar com objetos, especialmente para navegar e processar suas propriedades, incluindo objetos aninhados.

Exemplo Prático: Imprimindo Propriedades de um Objeto

Vamos criar uma função recursiva que percorre todas as propriedades de um objeto, incluindo objetos aninhados, e imprime suas chaves e valores.

Exemplo de objeto simples:

```
const carro = {  
    marca: "Toyota",  
    modelo: "Corolla",  
    ano: 2024,  
    pecas: {  
        portas: 4,  
        pneus: "Sistema Abs",  
        airbag: true,  
    },  
};
```



Função recursiva para imprimir propriedades:

```
function imprimirObjeto(objeto) {  
    // Obter as chaves do objeto  
    const keys = Object.keys(objeto); // Retorna um array com as chaves do objeto  
  
    // Caso base - se o objeto estiver vazio  
    if (keys.length === 0) {  
        return; // Encerra a função  
    }  
  
    // Iteração sobre as chaves do objeto  
    for (let i = 0; i < keys.length; i++) {  
        const key = keys[i];  
        const value = objeto[key];  
  
        // Verifica se o valor é um objeto  
        if (typeof value === "object") {  
            // Se for objeto, chama recursivamente a função  
            imprimirObjeto(value);  
        } else {  
            // Se não for objeto, imprime chave e seu valor  
            console.log(` ${key}: ${value}`);  
        }  
    }  
  
    // Chamada da função  
    imprimirObjeto(carro);  
}
```

Saída no console:

```
marca: Toyota  
modelo: Corolla  
ano: 2024  
portas: 4  
pneus: Sistema Abs  
airbag: true
```



Explicação do código:

- **Object.keys(objeto):**
 - Retorna um array com as chaves (propriedades) do objeto.
- **typeof:**
 - Verifica o tipo do valor associado a cada chave.
 - Caso o valor seja do tipo "object", a função é chamada novamente (recursão).
- **Caso base:**
 - Quando o array de chaves é vazio (objeto sem propriedades), a função para.

Outros métodos relacionados:

- **Object.values(objeto):**
 - Retorna um array com os valores das propriedades do objeto.
- **Object.entries(objeto):**
 - Retorna um array de pares [chave, valor] de um objeto.
- **Object.hasOwnProperty(propriedade):**
 - Verifica se o objeto possui uma propriedade específica (retorna true ou false).

Com esses métodos e o conceito de recursividade, é possível criar funções poderosas para manipular e analisar objetos de qualquer complexidade!



Fluxo da Função imprimirObjeto

- A função recebe um objeto como argumento.
- Utiliza o método **Object.keys(objeto)** para obter um array com as chaves do objeto.
- Para cada chave:
 - Obtém o valor associado.
 - Verifica se o valor é outro **objeto** usando `typeof value === "object"`.
- Caso o valor seja um objeto:
 - A função **se chama novamente** (recursivamente), passando o sub-objeto como argumento.
- Caso contrário:
 - Imprime a chave e o valor no console.
- O processo continua até que todos os objetos e sub-objetos sejam processados.

Onde está a Recursividade?

A recursividade está no trecho onde a função se chama novamente para processar sub-objetos:

```
if (typeof value === "object") {  
    imprimirObjeto(value); // Chamada recursiva  
}
```



For ... in

O **for...in** em JavaScript é uma estrutura de controle que permite percorrer as propriedades de um objeto. Ele é usado para iterar sobre as propriedades enumeráveis de um objeto, incluindo as propriedades herdadas.

A **sintaxe básica do for...in** é a seguinte:

```
for (variavel in objeto) {  
    // bloco de código a ser executado  
}
```

A cada iteração do loop, a variável especificada recebe o nome de uma propriedade do objeto. Você pode então usar essa variável para acessar o valor da propriedade correspondente.

É importante lembrar que o for...in não garante a ordem das propriedades percorridas. Além disso, ele também itera sobre as propriedades herdadas do objeto, o que pode não ser desejado em algumas situações.

Imagine um cenário no qual teremos um objeto com propriedades de um carro e aplicaremos o conceito de for...in nesse objeto. Vamos observar o que está ocorrendo no nosso programa:

```
JS iteratingOnObject.js > [?] car
1 const car = [
2   brand: 'Toyota',
3   model: 'Camry',
4   year: 2023,
5   color: 'Silver',
6   fuelType: 'Gasoline',
7   engineCapacity: '2.5L',
8   transmission: 'Automatic',
9   power: '203 horsepower',
10  seatingCapacity: 5,
11  price: '$25,000',
12 ];
13
14 for (info in car) {
15   console.log(info);
16 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL SQL CONSOLE

```
model
year
color
fuelType
engineCapacity
transmission
power
seatingCapacity
price
```

No código ao lado, temos um objeto chamado car que possui várias propriedades, como marca, modelo, ano, cor, tipo de combustível, capacidade do motor, transmissão, potência, capacidade de assentos e preço.

Em seguida, temos um loop for...in que percorre as propriedades do objeto car. A cada iteração do loop, a variável info recebe o nome de uma propriedade do objeto. Dentro do loop, temos a instrução console.log(info), que imprime no console o valor da variável info.

No caso do código fornecido, o console.log(info) irá imprimir no console o nome de cada propriedade do objeto car em cada iteração do loop.

Sso ocorre porque o for...in percorre todas as propriedades enumeráveis do objeto car, e a cada iteração, a variável info recebe o nome de uma propriedade. O console.log(info) imprime o valor da variável info, que é o nome da propriedade atual.

Dessa forma, o código está imprimindo no console o nome de cada propriedade do objeto car, permitindo visualizar todas as propriedades presentes no objeto.

```
JS iteratingOnObject.js > ...
1 const car = {
2   brand: 'Toyota',
3   model: 'Camry',
4   year: 2023,
5   color: 'Silver',
6   fuelType: 'Gasoline',
7   engineCapacity: '2.5L',
8   transmission: 'Automatic',
9   power: '203 horsepower',
10  seatingCapacity: 5,
11  price: '$25,000',
12 };
13
14 for (info in car) {
15   console.log(` ${info}: ${car[info]}`);
16 }
17
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL SQL CONSOLE

```
model: Camry
year: 2023
color: Silver
fuelType: Gasoline
engineCapacity: 2.5L
transmission: Automatic
power: 203 horsepower
seatingCapacity: 5
price: $25,000
```

No caso do exemplo de código ao lado, o `console.log({info}:{car[info]})` irá imprimir no console o nome de cada propriedade do objeto car, seguido por dois pontos e o valor da propriedade.

Isso ocorre porque, a cada iteração do loop, a variável info contém o nome de uma propriedade do objeto car. Usamos a notação de colchetes (`car[info]`) para acessar o valor da propriedade correspondente. Em seguida, usamos a interpolação de string (`${info}: ${car[info]}`) para criar uma string formatada que combina o nome da propriedade e o valor.

Dessa forma, o código está imprimindo no console o nome de cada propriedade do objeto car, seguido pelo seu valor correspondente, permitindo visualizar todas as propriedades e seus respectivos valores presentes no objeto.

Vamos analisar o comportamento do `for...in` quando executado para percorrer um Array:

No código ao lado, temos um array chamado `names` que contém uma lista de nomes. Em seguida, temos um loop `for...in` que percorre os índices do array `names`. A cada iteração do loop, a variável `personName` recebe o índice atual do array.

Dentro do loop, temos a instrução `console.log(personName)`, que imprime no console o valor da variável `personName`.

No caso do código fornecido, o `console.log(personName)` irá imprimir no console o índice de cada elemento do array `names` em cada iteração do loop. Portanto, o resultado no console será:

0
1
2
3
4
5
6
7
8

Isso ocorre porque o `for...in` percorre os índices do array `names`, e a cada iteração, a variável `personName` recebe o índice atual. O `console.log(personName)` imprime o valor da variável `personName`, que é o índice atual do array.

```
JS index.js > [o] names
1 const names = [
2   'Daniel',
3   'Izabelle',
4   'Fabrício',
5   'Rodrigo',
6   'Jorge',
7   'Lira',
8   'Camila',
9   'Matheus',
10  'Gabriel',
11];
```

No entanto, é importante destacar que o `for...in` não é a melhor opção para percorrer arrays em JavaScript. O `for...in` é mais adequado para percorrer as propriedades de um objeto. Para percorrer os elementos de um array, é recomendado utilizar o `for...of`.

For...of

O **for...of** em JavaScript é uma estrutura de controle que permite percorrer os elementos de uma coleção iterável, como um array, uma string, um objeto iterável ou um mapa. Ele é usado para executar uma determinada ação para cada elemento da coleção, sem a necessidade de acessar os índices ou propriedades.

A **sintaxe básica do for...of** é a seguinte:

```
for (variavel of colecao) {  
    // bloco de código a ser executado  
}
```

A cada iteração do loop, a variável especificada recebe o valor de um elemento da coleção. Você pode então usar essa variável para realizar alguma operação com o elemento atual.

O **for...of** é especialmente útil quando você precisa percorrer todos os elementos de uma coleção sem se preocupar com os índices ou propriedades. Ele simplifica o código e torna mais legível, pois você não precisa acessar os elementos usando índices ou métodos específicos.

O **for...of** também pode ser usado com outros tipos de coleções iteráveis, como strings, objetos iteráveis e mapas. Ele oferece uma maneira conveniente de percorrer os elementos de uma coleção sem a necessidade de lidar diretamente com os índices ou propriedades.



No código da aula anterior, substituímos o `for...in` pelo `for...of`. Agora, a variável `personName` irá receber diretamente o valor de cada elemento do array `names` a cada iteração do loop.

```
for (const personName of names) {  
    console.log(personName);  
}
```

O `console.log(personName)` irá imprimir no console o valor de cada elemento do array `names`.

```
Daniel  
Izabelle  
Fabrício  
Rodrigo  
Jorge  
Lira  
Camila  
Matheus  
Gabriel
```

Isso ocorre porque o `for...of` percorre os elementos do array `names`, e a cada iteração, a variável `personName` recebe o valor do elemento atual. O `console.log(personName)` imprime o valor da variável `personName`, que é o valor do elemento atual do array. O `for...of` é uma forma mais adequada de percorrer arrays em JavaScript, pois foi projetado especificamente para essa finalidade. Ele simplifica o código e torna mais claro o objetivo de percorrer os elementos do array.



Em JavaScript, existem valores que são considerados "**truthy**" (verdadeiros) e valores que são considerados "**falsy**" (falsos) em contextos booleanos. Esses valores são usados em expressões condicionais, como if e while, para determinar se uma condição é verdadeira ou falsa.

Valores "truthy" são aqueles que são considerados como verdadeiros quando avaliados em um contexto booleano. Isso significa que, se um valor "truthy" for usado em uma expressão condicional, ele será tratado como verdadeiro. Alguns exemplos de valores "truthy" em JavaScript são:

- **Strings não vazias:** qualquer string que contenha pelo menos um caractere é considerada "truthy".
- **Números diferentes de zero:** qualquer número diferente de zero é considerado "truthy".
- **Arrays e objetos:** mesmo que estejam vazios, arrays e objetos são considerados "truthy".
- **Funções:** qualquer função é considerada "truthy".

```
js index.js > ...
1 const variavelA = true;
2 if /*condição booleana*/ variavelA {
3   console.log('condição verdadeira');
4 }
```

```
/Users/danielporto/.nvm/versions/node/v20.3.1/bin/node ./index
condição verdadeira
```



Por outro lado, **valores "falsy"** são aqueles que são considerados como falsos quando avaliados em um contexto booleano. Isso significa que, se um valor "falsy" for usado em uma expressão condicional, ele será tratado como falso.

Alguns exemplos de **valores "falsy"** em JavaScript são:

- O valor false literal.
- O valor null.
- O valor undefined.
- O número 0.
- A string vazia "" (sem caracteres).
- O valor NaN (Not a Number).

```
const variavelFalsy = 0;
if (!variavelFalsy) {
    console.log('Executei porque a variável continha um valor com cara de falso.');
}
```

No código acima, temos uma constante chamada variavelFalsy que possui o valor 0.

Em seguida, temos uma instrução condicional if que verifica se a negação da variável variavelFalsy é verdadeira. A negação é feita usando o operador !.

Como o valor da variável variavelFalsy é 0, que é considerado "falsy" em JavaScript, a negação !variavelFalsy resulta em true. Portanto, o bloco de código dentro do if será executado. Nesse caso, o código irá imprimir no console a mensagem "Executei porque a variável continha um valor com cara de falso".

Isso ocorre porque quando a expressão condicional dentro do if é avaliada como verdadeira, o bloco de código dentro das chaves {} é executado. Como a negação de variavelFalsy é verdadeira, o código dentro do if é executado e a mensagem é exibida no console.

É importante notar que o valor 0 é considerado "falsy" em JavaScript, o que significa que é tratado como falso em contextos booleanos. Portanto, quando usamos a negação ! em variavelFalsy, obtemos true e o bloco de código dentro do if é executado.

Módulo 7

CONSOLIDANDO OS FUNDAMENTOS DO JAVASCRIPT

CONSOLIDANDO OS FUNDAMENTOS DO JAVASCRIPT

CONSOLIDANDO OS FUNDAMENTOS DO JAVASCRIPT

LABORATÓRIO DE FUNDAMENTOS

Parabéns por chegar até aqui! Você já explorou conceitos fundamentais para dominar JavaScript, como recursividade, manipulação de objetos e métodos essenciais. Agora é o momento de consolidar seu aprendizado por meio de exercícios práticos. Os exercícios a seguir foram cuidadosamente projetados para reforçar os conceitos apresentados até o momento. Cada um deles inclui exemplos de **entradas** e **saídas** esperadas, o que ajudará você a compreender o objetivo de cada atividade e validar suas soluções.

Como aproveitar ao máximo os exercícios?

- **Leia atentamente o enunciado:** Certifique-se de entender o problema antes de começar a resolvê-lo.
- **Analise os exemplos fornecidos:** Use os exemplos de entrada e saída como guias para entender o comportamento esperado do código.
- **Teste suas soluções:** Aplique diferentes entradas para verificar se o código funciona em diversos cenários.
- **Reflita sobre o aprendizado:** Ao finalizar um exercício, pense em como ele se conecta aos conceitos estudados e o que você pode melhorar na sua abordagem.

Estes exercícios não apenas reforçam seu entendimento, mas também desenvolvem suas habilidades para resolver problemas e escrever códigos mais eficientes e elegantes.

Prepare-se para colocar em prática o que aprendeu e aproveitar essa oportunidade de crescer como programador!

1. Função: somar

Descrição: Uma função simples que recebe dois números inteiros e retorna a soma deles.

Conceitos utilizados:

- **Função:** Definimos uma função utilizando function, que é uma unidade de código reutilizável.
- **Parâmetros:** x e y são os valores fornecidos à função para que ela realize o cálculo.
- **Operação aritmética:** A soma é realizada com o operador +.
- **Retorno:** A função usa return para enviar o resultado da operação.

Resolução:

```
function somar(x, y) {  
    let resultado = x + y; // Soma os dois valores recebidos  
    return resultado; // Retorna o valor da soma  
}
```

Entrada e saída:

```
console.log(somar(5, 3)); // Entrada: 5, 3. Saída: 8
```

2. Função: ordenarNomes

Descrição: A função recebe um array de nomes e retorna o array ordenado em ordem alfabética.

Conceitos utilizados:

- **Função:** Criada para encapsular a lógica de ordenação.
- **Array:** Uma estrutura de dados que armazena múltiplos valores.
- **Método sort:** É usado para ordenar os elementos de um array em ordem crescente. Por padrão, sort organiza os valores em ordem alfabética para strings.
- **Retorno:** O array ordenado é retornado diretamente.

Resolução:

```
function ordenarNomes(listaNomes) {  
    return listaNomes.sort(); // Ordena os nomes em ordem alfabética  
}
```

Entrada e saída:

```
let nomes = ["Zeus", "Ana", "Elias", "Carlos", "Beatriz"];  
console.log(ordenarNomes(nomes)); // Entrada: ["Zeus", "Ana", "Elias", "Carlos", "Beatriz"]. Saída: ["Ana", "Beatriz", "Carlos", "Elias", "Zeus"]
```

3. Função: compararValores

Descrição: A função verifica se dois valores fornecidos são considerados **truthy**. Retorna true se ambos forem verdadeiros e false caso contrário.

Conceitos utilizados:

- **Truthy e Falsy:** Valores como 0, null, undefined, false e "" (string vazia) são falsy, enquanto outros valores são truthy.
- **Operador lógico &&:** Retorna true apenas se ambos os operandos forem verdadeiros.
- **Função Boolean:** Converte um valor para seu equivalente lógico (verdadeiro ou falso).
- **Retorno:** O resultado da comparação lógica é retornado.

Resolução:

```
function compararValores(valor1, valor2) {  
    let booleano = Boolean(valor1 && valor2); // Verifica se ambos os valores são truthy  
    return booleano;  
}
```

Entrada e saída:

```
console.log(compararValores(5, "texto")); // true - Ambos são truthy  
console.log(compararValores(0, "texto")); // false - 0 é falsy  
console.log(compararValores(0, false)); // false - Ambos são falsy
```



4. Função: calcularRendimento

Descrição: A função calcula o rendimento obtido a partir de um valor investido e uma taxa de juros anual.

Conceitos utilizados:

- **Função:** Encapsula a lógica de cálculo.
- **Parâmetros:** Recebe o valor investido e a taxa de juros como argumentos.
- **Cálculo percentual:** A fórmula $\text{valor} * (\text{taxa} / 100)$ é usada para calcular a porcentagem.
- **Retorno:** O valor calculado do rendimento é retornado.

Resolução:

```
function calcularRendimento(valorInvestido, taxaJuros) {  
    let rendimento = valorInvestido * (taxaJuros / 100); // Calcula o rendimento com base na taxa  
    return rendimento;  
}
```

Entrada e saída:

```
console.log(calcularRendimento(1000, 5)); // Entrada: 1000 (valor), 5 (taxa). Saída: 50 (rendimento)
```

Resumo dos Conceitos Reforçados

- **Funções:** Estruturas que encapsulam lógicas e operações reutilizáveis.
- **Parâmetros e Retorno:** Permitem fornecer entradas às funções e recuperar saídas.
- **Arrays e Métodos:** Uso do método sort para ordenar arrays.
- **Lógica Booleana:** Avaliação de valores truthy e falsy com Boolean e operadores lógicos.
- **Cálculos Matemáticos:** Aplicação de fórmulas para resolver problemas práticos.

Esse exercícios ajudam a consolidar a base do JavaScript, abordando operações matemáticas, manipulação de arrays, lógica booleana e cálculos.

5. Função: dividirFrase

Objetivo: Separar uma frase em palavras, usando espaços como delimitadores, e retornar essas palavras em um array.

Conceitos:

- A função **split()** divide uma string em um array, de acordo com o delimitador especificado (neste caso, o espaço " ").
- O **return** retorna o array resultante.

Resolução:

```
function dividirFrase(frase) {  
  let fraseDivida = frase.split(" ");  
  return fraseDivida;  
}
```

Entrada e saída:

```
let frase = "Esta é uma frase de exemplo.";  
console.log(dividirFrase(frase)); // Saída: ["Esta", "é", "uma", "frase", "de", "exemplo."]
```



6. Função: calcularPontos

Objetivo: Calcular a quantidade total de pontos de um time, sabendo que uma vitória vale 3 pontos e um empate vale 1 ponto.

Conceitos:

- A função recebe dois parâmetros: o número de vitórias e empates.
- Multiplicamos o número de vitórias por 3 (já que cada vitória vale 3 pontos).
- Somamos os empates (já que cada empate vale 1 ponto).

Resolução:

```
function calcularPontos(vitorias, empates) {  
    return `${vitorias * 3 + empates} pontos`;  
}
```

Entrada e Saída:

```
console.log(calcularPontos(5, 2)); // Saída: "17 pontos"
```

7. Função: contarVogais

Objetivo: Contar o número de vogais presentes em uma frase.

Conceitos:

- Utilizamos um **loop** para iterar sobre cada caractere da frase.
- A variável vogais armazena todas as vogais, incluindo acentuadas e com til.
- O método **includes()** verifica se a letra atual é uma vogal.
- O **contador** é incrementado sempre que encontramos uma vogal.

Resolução:

```
function contarVogais(frase) {  
    let vogais = "aeiouáéíóúãâêîôû";  
    let contador = 0;  
    for (let index = 0; index < frase.length; index++) {  
        const letra = frase[index].toLowerCase();  
        if (vogais.includes(letra)) {  
            contador++;  
        }  
    }  
    return contador;  
}
```

Entrada e Saída:

```
console.log(contarVogais("Olá, tudo bem úia?")); // Saída: 5
```



8. Função: maiorGasto

Descrição: A função encontra o maior valor de gasto em um array e conta quantas vezes esse maior gasto se repete.

Conceitos utilizados:

- **Looping:** Para percorrer o array e encontrar o maior valor.
- **Condicional:** Para atualizar o maior valor encontrado e contar as repetições desse valor.
- **Variáveis de controle:** Para armazenar o maior valor e o contador das repetições.

Resolução:

```
function maiorGasto(gastos) {
    let contador = 0;
    let maiorGastoAtual = gastos[0];

    for (let index = 0; index < gastos.length; index++) {
        if (gastos[index] > maiorGastoAtual) {
            maiorGastoAtual = gastos[index];
        }
    }

    for (let i = 0; i < gastos.length; i++) {
        if (gastos[i] === maiorGastoAtual) {
            contador++;
        }
    }
    return `O maior gasto é de ${maiorGastoAtual} e ele repete ${contador} vezes`;
}
```

Entrada e Saída:

```
let entrada = [10, 5, 20, 15, 20, 11, 20];
console.log(maiorGasto(entrada)); // Entrada: [10, 5, 20, 15, 20, 11, 20]. Saída: "O maior gasto é de 20 e ele repete 3 vezes"
```



9. Função: calcularMedia

Descrição: A função calcula a média dos números em um array.

Conceitos utilizados:

- **Looping:** Para somar todos os valores do array.
- **Operações matemáticas:** Para calcular a soma e dividir pela quantidade de elementos para obter a média.

Resolução:

```
function calcularMedia(numeros) {  
    let soma = 0;  
    for (let index = 0; index < numeros.length; index++) {  
        soma += numeros[index];  
    }  
    return soma / numeros.length;  
}
```

Entrada e saída:

```
let numerosLista = [10, 20, 30, 40];  
console.log(calcularMedia(numerosLista)); // Entrada: [10, 20, 30, 40]. Saída: 25
```



10. Função: ehPalindromo

Descrição: A função verifica se uma palavra é um palíndromo (uma palavra que pode ser lida de trás para frente e de frente para trás, mantendo o mesmo significado).

Conceitos utilizados:

- **String:** Manipulação de strings para inverter a palavra.
- **Looping:** Para percorrer a palavra de trás para frente e construir a palavra invertida.
- **Comparação:** Verificação de igualdade entre a palavra original e a invertida.

Resolução:

```
function ehPalindromo(palavra) {  
    let palavraInvertida = "";  
    for (let index = palavra.length - 1; index >= 0; index--) {  
        palavraInvertida += palavra[index];  
    }  
    let ehPalindromo = palavra === palavraInvertida;  
    return ehPalindromo;  
}
```

Entrada e saída:

```
console.log(ehPalindromo("arara")); // Entrada: "arara" Saída: true  
console.log(ehPalindromo("ana")); // Entrada: "ana" Saída: true  
console.log(ehPalindromo("cachorro")); // Entrada: "cachorro" Saída: false
```



11. Função: ehPrimo

Descrição: A função verifica se um número é primo (um número maior que 1 que só é divisível por 1 e por ele mesmo).

Conceitos utilizados:

- **Condisional:** Para verificar se o número é menor ou igual a 1 (não é primo).
- **Looping:** Para verificar se o número é divisível por qualquer outro número menor que ele.
- **Operações matemáticas:** Para verificar divisibilidade.

Resolução:

```
function ehPrimo(numero) {  
    if (numero <= 1) return false;  
    for (let index = 2; index < numero; index++) {  
        if (numero % index === 0) return false;  
    }  
    return true;  
}
```

Entrada e saída:

```
console.log(ehPrimo(7)); // Entrada: 7 Saída: true  
console.log(ehPrimo(4)); // Entrada: 4 Saída: false
```



12. Função: contarPalavras

Descrição: A função conta a quantidade de palavras em uma frase.

Conceitos utilizados:

- **String:** Manipulação de strings para dividir a frase em palavras.
- **Método split():** Para dividir a frase com base nos espaços em branco.
- **Looping:** Para contar o número de palavras, ignorando as vazias.

Resolução:

```
function contarPalavras(frase) {
    let contador = 0;
    const palavras = frase.trim().split(" "); // Divide a frase em palavras e remove espaços extras

    for (let index = 0; index < palavras.length; index++) {
        if (palavras[index] !== "") { // Ignora palavras vazias
            contador++;
        }
    }
    return contador;
}
```

Entrada e saída:

```
let frase = "Olá, tudo bem com você?";
console.log(contarPalavras(frase)); // Entrada: "Olá, tudo bem com você?" Saída: 4
```



13. Função: calcularDistancia

Descrição: A função compara duas distâncias e retorna qual pessoa está mais próxima. Se a primeira distância for menor, retorna "Pessoa 1", se a segunda for menor, retorna "Pessoa 2", e se ambas forem iguais, retorna "Ambos estão a mesma distância".

Conceitos utilizados

- **Condicional (if / else if / else):** Utilizado para verificar qual das distâncias é menor e retornar a resposta adequada.
- **Comparação (<, >, ===):** Verifica a relação entre os valores das distâncias.
- **Retorno (return):** Garante que a função devolva o resultado correto com base na comparação realizada.

Resolução:

```
function calcularDistancia(distancia1, distancia2) {  
    if (distancia1 < distancia2) {  
        return "Pessoa 1";  
    } else if (distancia1 > distancia2) {  
        return "Pessoa 2";  
    } else {  
        return "Ambos estão a mesma distância";  
    }  
}
```

Entrada e saída:

```
console.log(calcularDistancia(10, 15)); // Entrada: (10, 15). Saída: "Pessoa 1"  
console.log(calcularDistancia(20, 15)); // Entrada: (20, 15). Saída: "Pessoa 2"  
console.log(calcularDistancia(30, 30)); // Entrada: (30, 30). Saída: "Ambos estão a mesma distância"
```



14. Função: controleDespesas

Descrição: A função recebe um array de despesas e retorna um novo array classificando cada despesa como "Alto Gasto" se for maior que 100, ou "Gasto Controlado" se for menor ou igual a 100.

Conceitos utilizados

- **Array ([])**: A estrutura de dados utilizada para armazenar as despesas e o resultado da classificação.
- **Laço de repetição (for...of)**: Percorre os elementos do array de despesas.
- **Condisional (if / else)**: Verifica se a despesa é maior que 100 para classificá-la corretamente.
- **Método push()**: Adiciona elementos ao array de resultado.

Resolução:

```
function controleDespesas(despesas) {  
    const resultado = [];  
    for (let despesa of despesas) {  
        if (despesa > 100) {  
            resultado.push("Alto Gasto");  
        } else {  
            resultado.push("Gasto Controlado");  
        }  
    }  
    return resultado;  
}  
  
// Exemplo de uso  
const despesas = [150, 80, 200, 60, 120];  
console.log(controleDespesas(despesas));
```

Entrada e saída:

```
const despesas = [150, 80, 200, 60, 120];  
console.log(controleDespesas(despesas));  
// Entrada: [150, 80, 200, 60, 120]  
// Saída: ["Alto Gasto", "Gasto Controlado", "Alto Gasto", "Gasto Controlado", "Alto Gasto"]
```



15.Função: gerarListaInvestimentos

Descrição: A função recebe um array de valores de investimentos e um nome. Ela retorna um novo array de objetos, onde cada objeto contém o valor do investimento e o nome fornecido. Além disso, os investimentos são ordenados do menor para o maior.

Conceitos utilizados

- **Condicional (if):** Verifica se o array está vazio e retorna "Vazio!" se não houver investimentos.
- **Array e objetos ([] e {}):** O array armazena os objetos que contêm as informações dos investimentos.
- **Laço de repetição (for...of):** Percorre os elementos do array de investimentos.
- **Método push():** Adiciona objetos ao array resultante.
- **Ordenação (sort()):** Organiza os investimentos em ordem crescente.

Resolução:

```
function gerarListaInvestimentos(investimentos, nome) {  
  if (investimentos.length === 0) {  
    return "Vazio!";  
  }  
  
  const lista = [];  
  for (let investimento of investimentos) {  
    lista.push({ investimento: investimento, nome: nome });  
  }  
  
  lista.sort((a, b) => (a.investimento > b.investimento ? 1 : -1)); // Ordena os investimentos em ordem crescente  
  
  return lista;  
}  
  
// Exemplo de uso  
const investimentos = [5000, 2000, 15000];  
const nome = "Maria";  
console.log(gerarListaInvestimentos(investimentos, nome));
```

Entrada e saída:

```
const investimentos = [5000, 2000, 15000];  
const nome = "Maria";  
console.log(gerarListaInvestimentos(investimentos, nome));  
// Entrada: ([5000, 2000, 15000], "Maria")  
// Saída:  
// [  
//   { investimento: 2000, nome: "Maria" },  
//   { investimento: 5000, nome: "Maria" },  
//   { investimento: 15000, nome: "Maria" }  
// ]
```



16. Funções: codificar e decodificar

Descrição: As funções **codificar** e **decodificar** servem para transformar frases, substituindo vogais por números e revertendo essa substituição.

- **Na função codificar**, as vogais são convertidas conforme a regra:
 - a → 1
 - e → 2
 - i → 3
 - o → 4
 - u → 5
- **Na função decodificar**, o processo é revertido para recuperar a frase original.

Conceitos utilizados

- **Laço while e replace()**
 - O método `.replace("a", "l")` substitui a primeira ocorrência da letra "a" por "l".
 - Para substituir todas as ocorrências, usamos um while que continua até não haver mais a letra a ser substituída.
- **Expressões Regulares (RegExp)**
 - Utilizamos `replace(/a/g, "l")`, onde `/a/g` significa:
 - `/a/` → Procurar a letra "a".
 - `g` (global) → Substituir todas as ocorrências.
- **Refatoração**
 - O código inicial com while pode ser otimizado usando **expressões regulares**, reduzindo a quantidade de linhas.



Passo a passo da solução

- Versão com while e replace()

```
function codificar(frase) {
  let resultado = frase;

  while (resultado.includes("a")) {
    resultado = resultado.replace("a", "1");
  }

  while (resultado.includes("e")) {
    resultado = resultado.replace("e", "2");
  }

  while (resultado.includes("i")) {
    resultado = resultado.replace("i", "3");
  }

  while (resultado.includes("o")) {
    resultado = resultado.replace("o", "4");
  }

  while (resultado.includes("u")) {
    resultado = resultado.replace("u", "5");
  }

  return resultado;
}
```

```
function decodificar(frase) {
  let resultado = frase;

  while (resultado.includes("1")) {
    resultado = resultado.replace("1", "a");
  }

  while (resultado.includes("2")) {
    resultado = resultado.replace("2", "e");
  }

  while (resultado.includes("3")) {
    resultado = resultado.replace("3", "i");
  }

  while (resultado.includes("4")) {
    resultado = resultado.replace("4", "o");
  }

  while (resultado.includes("5")) {
    resultado = resultado.replace("5", "u");
  }

  return resultado;
}
```

Entrada e Saída:

```
// Testes
const fraseCodificar = "a casa e o sol um dia lindo";
const resultadoCodificado = codificar(fraseCodificar);
console.log(resultadoCodificado); // "1 c1s1 2 4 s4l 5m d31 l3nd4"

console.log(decodificar(resultadoCodificado)); // "a casa e o sol um dia lindo"
```

```
const frase = "a casa e o sol um dia lindo";
console.log(codificar(frase));
// Entrada: "a casa e o sol um dia lindo"
// Saída: "1 c1s1 2 4 s4l 5m d31 l3nd4"

console.log(decodificar("1 c1s1 2 4 s4l 5m d31 l3nd4"));
// Entrada: "1 c1s1 2 4 s4l 5m d31 l3nd4"
// Saída: "a casa e o sol um dia lindo"
```



- **Versão otimizada com Expressões Regulares (RegExp)**

Essa versão substitui todas as vogais ao mesmo tempo usando .replace() encadeado.

```
function codificarRegex(frase) {  
    return frase  
        .replace(/a/g, "1")  
        .replace(/e/g, "2")  
        .replace(/i/g, "3")  
        .replace(/o/g, "4")  
        .replace(/u/g, "5");  
  
}  
  
function decodificarRegex(frase) {  
    return frase  
        .replace(/1/g, "a")  
        .replace(/2/g, "e")  
        .replace(/3/g, "i")  
        .replace(/4/g, "o")  
        .replace(/5/g, "u");  
}
```

Entrada e Saída:

```
// Testes  
console.log(codificarRegex(fraseCodificar)); // "1 c1s1 2 4 s4l 5m d31 l3nd4"  
console.log(decodificarRegex(resultadoCodificado)); // "a casa e o sol um dia lindo"
```

```
const frase = "a casa e o sol um dia lindo";  
console.log(codificar(frase));  
// Entrada: "a casa e o sol um dia lindo"  
// Saída: "1 c1s1 2 4 s4l 5m d31 l3nd4"  
  
console.log(decodificar("1 c1s1 2 4 s4l 5m d31 l3nd4"));  
// Entrada: "1 c1s1 2 4 s4l 5m d31 l3nd4"  
// Saída: "a casa e o sol um dia lindo"
```



Comparação entre as versões

Método	Código mais curto	Desempenho melhor	Facilidade de entender
<code>while + replace()</code>	✗ Não	✗ Não (executa múltiplos loops)	✓ Sim
Expressões Regulares <code>(RegExp)</code>	✓ Sim	✓ Sim (substitui tudo de uma vez)	✗ Pode ser confuso

Para quem está começando, a versão com while é mais intuitiva, mas a versão com **expressões regulares** é **mais eficiente e elegante**.

Conclusão: Se o objetivo é otimizar código, prefira a versão com **RegExp**.

LABORATÓRIO DE SIMULAÇÃO FINANCEIRA

Parabéns por chegar até aqui! Você já explorou conceitos fundamentais de programação em JavaScript, como manipulação de objetos, criação de funções e estruturação de dados. Agora é o momento de consolidar esse conhecimento por meio de um projeto prático de **Simulação Financeira**.

Os desafios a seguir foram cuidadosamente elaborados para que você possa aplicar e reforçar conceitos essenciais. Durante a implementação, você trabalhará com cálculos financeiros, gerenciamento de despesas e geração de relatórios, utilizando abordagens que refletem cenários reais.

Como aproveitar ao máximo este projeto?

- **Leia atentamente os requisitos:** Antes de iniciar a implementação, compreenda cada funcionalidade exigida e sua finalidade dentro do projeto.
- **Estruture bem seus dados:** Utilize objetos e arrays de maneira organizada para facilitar a manipulação das informações.
- **Teste suas funções com diferentes entradas:** Simule diversos cenários para garantir que seu código se adapta a diferentes situações e regras de negócio.
- **Refatore e melhore:** Após validar suas soluções, reflita sobre como otimizar seu código, tornando-o mais eficiente e legível.
- Este projeto não apenas reforçará seu entendimento sobre programação, mas também desenvolverá sua capacidade de solucionar problemas e estruturar códigos mais organizados.

Agora é a hora de colocar a mão no código e aprimorar suas habilidades como desenvolvedor!

1. Função: simularInvestimento

Descrição: A função simularInvestimento calcula o saldo final de um investimento com base em um valor inicial, uma taxa mensal de rendimento e um período determinado em meses. O cálculo é baseado no **rendimento composto**, onde o saldo acumulado é atualizado a cada mês.

Conceitos utilizados:

- **Laço de repetição (for):** Utilizado para iterar ao longo do período de meses e calcular o rendimento acumulado.
- **Cálculo de juros compostos:** A cada iteração, o saldo recebe um acréscimo proporcional à taxa mensal.
- **Operação matemática com porcentagem:** A taxa de rendimento é aplicada dividindo o valor por 100.
- **Formatação de números (toFixed(2)):** Para garantir que o resultado tenha duas casas decimais.

Resolução:

```
function simularInvestimento(valorInicial, taxaMensal, meses) {  
    let saldoFinal = valorInicial; // Define o saldo inicial  
  
    for (let index = 1; index <= meses; index++) {  
        saldoFinal += saldoFinal * (taxaMensal / 100); // Aplica a taxa de rendimento composta  
    }  
  
    return saldoFinal.toFixed(2); // Retorna o saldo final com duas casas decimais  
}
```

Entrada e saída:

```
console.log(simularInvestimento(1000, 3, 6));
// Entrada: 1000 (valor inicial), 3% (taxa mensal), 6 meses
// Saída: "1195.62" (saldo final após os meses de rendimento)

console.log(simularInvestimento(500, 5, 12));
// Entrada: 500 (valor inicial), 5% (taxa mensal), 12 meses
// Saída: "895.42"

console.log(simularInvestimento(1500, 2, 4));
// Entrada: 1500 (valor inicial), 2% (taxa mensal), 4 meses
// Saída: "1624.32"
```

Passo a passo da execução:

- 1 **Definição do saldo inicial** → O valor do investimento começa com valorInicial.
- 2 **Iteração com for** → Para cada mês, o saldo é atualizado aplicando a taxa de rendimento.
- 3 **Cálculo dos juros compostos** → O saldo é multiplicado pela taxa e somado ao próprio saldo.
- 4 **Retorno do valor formatado** → O resultado é ajustado para exibição com duas casas decimais.

Essa função permite simular diferentes cenários financeiros e pode ser utilizada para análises de investimentos!



2.Função: gerenciarDespesas

Descrição: A função gerenciarDespesas recebe um objeto contendo despesas categorizadas e calcula o **total gasto** somando os valores de cada categoria.

Conceitos utilizados:

- **Objetos (Object):** Utilizados para armazenar as despesas em diferentes categorias.
- **Laço for...in:** Itera sobre as propriedades do objeto, acessando cada categoria.
- **Acesso dinâmico às propriedades do objeto:** Utilizando despesas[categoria] para obter os valores.
- **Acumulação de valores:** A cada iteração, os valores das despesas são somados para obter o total.

Resolução:

```
function gerenciarDespesas(despesas) {  
    let totalDespesas = 0; // Inicializa a variável que armazenará o total de despesas  
  
    for (let categoria in despesas) {  
        totalDespesas += despesas[categoria]; // Soma os valores de cada categoria  
    }  
  
    return totalDespesas; // Retorna o total das despesas  
}
```



Entrada e saída:

```
const despesas = {
    alimentacao: 500,
    transporte: 300,
    aluguel: 1000,
    lazer: 250,
};

console.log(gerenciarDespesas(despesas));
// Entrada: { alimentacao: 500, transporte: 300, aluguel: 1000, lazer: 250 }
// Saída: 2050
```

Passo a passo da execução:

- 1 **Criação do objeto despesas** → Contém categorias como alimentação, transporte, aluguel e lazer.
- 2 **Inicialização do total** → Define totalDespesas = 0 para armazenar a soma.
- 3 **Iteração com for...in** → Percorre todas as categorias do objeto e soma os valores.
- 4 **Retorno do total** → A função retorna a soma total das despesas.

Essa função é útil para análise financeira, permitindo calcular rapidamente os gastos mensais em diferentes categorias!

3. Função: obterMesAtual

Descrição: A função obterMesAtual retorna o nome do mês atual baseado na data do sistema.

Conceitos utilizados:

- **Objeto Date**: Permite trabalhar com datas e horários no JavaScript.
- **Método .getMonth()**: Retorna o número do mês (de **0 a 11**).
- **Arrays**: Utilizado para armazenar os nomes dos meses.
- **Acesso a elementos do array**: O índice do mês retornado pelo .getMonth() é usado para buscar o nome correto.

Resolução:

```
function obterMesAtual() {  
    const meses = [  
        "Janeiro", "Fevereiro", "Março", "Abril", "Maio", "Junho",  
        "Julho", "Agosto", "Setembro", "Outubro", "Novembro", "Dezembro"  
    ];  
  
    const dataAtual = new Date(); // Obtém a data atual do sistema  
    return meses[dataAtual.getMonth()]; // Retorna o nome do mês correspondente ao índice  
}
```



Entrada e saída:

```
console.log(obterMesAtual());
// Se executado em fevereiro -> Saída: "Fevereiro"
// Se executado em outubro -> Saída: "Outubro"
```

Passo a passo da execução:

- 1 **Criação do array meses** → Contém os nomes dos meses, de janeiro (índice 0) a dezembro (índice 11).
- 2 **Criação do objeto Date** → new Date() obtém a data e horário atual do sistema.
- 3 **Uso do .getMonth()** → Retorna o número do mês atual, de 0 a 11.
- 4 **Acesso ao array meses** → O índice retornado é utilizado para buscar o nome correto.

Essa função é útil para relatórios financeiros, geração de datas dinâmicas e qualquer aplicação que necessite exibir o mês atual de forma legível!



4.Função: gerarRelatorio (parte 1)

Descrição: A função gerarRelatorio gera um relatório financeiro com informações sobre o investimento, despesas e o mês atual.

Conceitos utilizados:

- **Reutilização de funções:** Chama outras funções já criadas (simularInvestimento, gerenciarDespesas e obterMesAtual).
- **Interpolação de strings:** Usa **template literals** (` `) para exibir valores no console.
- **Organização do código:** Estrutura clara e modularizada.

Resolução:

```
function gerarRelatorio(
    investimentoInicial,
    taxa,
    meses,
    despesas,
    metaInvestimento,
    metaOrcamento
) {
    const saldoInvestimento = simularInvestimento(investimentoInicial, taxa, meses);
    const totalDespesas = gerenciarDespesas(despesas);
    const mesAtual = obterMesAtual();

    console.log("== Relatório Financeiro ==");
    console.log(`Mês: ${mesAtual}`);
    console.log(`Saldo final do investimento: R$ ${saldoInvestimento}`);
    console.log(`Total de despesas: R$ ${totalDespesas}`);
}
```



Entrada e saída:

```
const despesas = {  
    alimentação: 500,  
    transporte: 300,  
    aluguel: 1000,  
    lazer: 250  
};  
  
gerarRelatorio(1000, 5, 6, despesas, 5000, 2000);
```

Saída esperada no console:

```
==== Relatório Financeiro ====  
Mês: Fevereiro  
Saldo final do investimento: R$ 1340.10  
Total de despesas: R$ 2050
```

Passo a passo da execução:

- 1 **Calcula o saldo do investimento** → Usa simularInvestimento para obter o valor final do investimento.
- 2 **Calcula o total das despesas** → Usa gerenciarDespesas para somar todos os gastos.
- 3 **Obtém o mês atual** → Usa obterMesAtual para exibir o nome correto.
- 4 **Exibe os dados no console** → Utiliza console.log para formatar o relatório.

Essa primeira parte do relatório estabelece a base para exibir as informações financeiras essenciais. A próxima etapa incluirá a comparação com metas e uma análise mais detalhada!



4.Função: gerarRelatorio (Parte 2)

Descrição:

A segunda parte da função gerarRelatorio expande a análise financeira, incluindo:

- **Cálculo do orçamento** → Verifica se houve economia ou excedente.
- **Meta de investimento** → Confere se o saldo final atingiu a meta estabelecida.
- **Resumo detalhado das despesas** → Exibe os gastos por categoria.

Resolução:

```
function gerarRelatorio(
  investimentoInicial,
  taxa,
  meses,
  despesas,
  metaInvestimento,
  metaOrcamento
) {
  const saldoInvestimento = simularInvestimento(investimentoInicial, taxa, meses);
  const totalDespesas = gerenciarDespesas(despesas);
  const mesAtual = obterMesAtual();

  console.log("== Relatório Financeiro ==");
  console.log(`Mês: ${mesAtual}`);
  console.log(`Saldo final do investimento: R$ ${saldoInvestimento}`);
  console.log(`Total de despesas: R$ ${totalDespesas}`);

  // ✨ Orçamento: verificar se houve economia ou excedente
  const economia = metaOrcamento - totalDespesas;
  if (economia > 0) {
    console.log(`Parabéns! Você economizou R$ ${economia}`);
  } else {
    console.log(`Você excedeu seu orçamento em R$ ${Math.abs(economia)}`);
  }

  // ✨ Meta de investimento: atingida ou não?
  if (saldoInvestimento >= metaInvestimento) {
    console.log("Meta de investimento atingida!");
  } else {
    console.log("Você não atingiu a meta de investimento");
  }

  // ✨ Resumo das despesas por categoria
  console.log("== Resumo das Despesas ==");
  for (const categoria in despesas) {
    console.log(
      `${categoria.charAt(0).toUpperCase() + categoria.slice(1)}: R$ ${despesas[categoria]}`;
    );
  }
}
```

Entrada e saída:

```
const despesas = {  
    alimentação: 500,  
    transporte: 300,  
    aluguel: 1000,  
    lazer: 250  
};  
  
gerarRelatorio(1000, 5, 6, despesas, 5000, 2000);
```

Saída esperada no console:

```
== Relatório Financeiro ==  
Mês: Fevereiro  
Saldo final do investimento: R$ 1340.10  
Total de despesas: R$ 2050  
Você excedeu seu orçamento em R$ 50  
Você não atingiu a meta de investimento  
== Resumo das Despesas ==  
Alimentação: R$ 500  
Transporte: R$ 300  
Aluguel: R$ 1000  
Lazer: R$ 250
```

Passo a passo da execução:

- 1 Calcula a economia ou excesso de gastos** → metaOrcamento - totalDespesas
 - Se for positivo, exibe uma mensagem de economia.
 - Se for negativo, informa o excedente.
- 2 Verifica a meta de investimento** → Compara saldoInvestimento com metaInvestimento.
 - Se o saldo for maior ou igual, a meta foi atingida.
 - Caso contrário, informa que a meta não foi alcançada.
- 3 Lista as despesas por categoria** → Itera sobre o objeto despesas e formata os valores.

Execução do Relatório com Entradas e Configurações do Investimento

Objetivo:

Aqui, estamos utilizando os requisitos anteriores para gerar um relatório financeiro completo, com todas as configurações do investimento e as despesas mensais definidas. Vamos ver o que cada parte do código faz:

Entradas:

- **Despesas:** Um objeto com categorias de despesas e seus respectivos valores. Cada categoria possui um valor associado, como por exemplo, o condomínio (R\$ 800), escola (R\$ 500), etc.

```
const despesas = {  
    condominio: 800,  
    escola: 500,  
    academia: 150,  
    agua: 60,  
    luz: 120,  
    lazer: 300,  
};
```

- **Configurações do Investimento:** Variáveis que definem os parâmetros do investimento, como o valor inicial, a taxa mensal, a duração em meses, e as metas de investimento e orçamento mensal.

```
const valorInicial = 3000; // Valor inicial investido
const taxaMensal = 1.3; // Taxa de rendimento mensal em %
const meses = 12; // Duração do investimento
const metaInvestimento = 2000; // Meta para o saldo final do investimento
const metaOrcamento = 1500; // Meta de orçamento mensal
```

Execução do Relatório:

Ao chamar a função gerarRelatorio(), estamos passando todas as variáveis e objetos como parâmetros. A função irá processar esses dados e gerar um relatório completo com as seguintes informações:

- **Mês Atual:** A função irá identificar o mês atual com base na data e exibi-lo no relatório.
- **Saldo Final do Investimento:** Calcula o saldo final após o período do investimento, levando em consideração a taxa de rendimento.
- **Total de Despesas:** Soma de todas as despesas mensais informadas no objeto despesas.
- **Economia ou Excedente:** Compara o total de despesas com a meta de orçamento e exibe uma mensagem de economia ou excedente.
- **Meta de Investimento:** Verifica se a meta de saldo final do investimento foi atingida ou não.
- **Resumo das Despesas por Categoria:** Exibe uma listagem de todas as despesas por categoria com seus respectivos valores.

Execução do Código:

```
gerarRelatorio(  
    valorInicial,  
    taxaMensal,  
    meses,  
    despesas,  
    metaInvestimento,  
    metaOrcamento  
) ;
```

Saída Esperada no Console:

Com as entradas fornecidas, o código gerará um relatório como o seguinte (a saída pode variar dependendo do mês atual):

```
==== Relatório Financeiro ====  
Mês: Fevereiro  
Saldo final do investimento: R$ 3473.40  
Total de despesas: R$ 2030  
Você excedeu seu orçamento em R$ 530  
Você não atingiu a meta de investimento  
==== Resumo das Despesas ====  
Condomínio: R$ 800  
Escola: R$ 500  
Academia: R$ 150  
Água: R$ 60  
Luz: R$ 120  
Lazer: R$ 300
```



Explicação dos Resultados:

- **Saldo final do investimento:** A função simularInvestimento calcula o saldo final com base no valor inicial (R\$ 3000), taxa mensal de 1.3% e duração de 12 meses.

Após 12 meses, o saldo final do investimento será de R\$ 3473.40.

- **Total de despesas:** O valor total das despesas foi somado com base no objeto despesas fornecido. O total é R\$ 2030.
- **Economia ou Excedente:** A diferença entre a meta de orçamento de R\$ 1500 e o total de despesas de R\$ 2030 resulta em um excedente de R\$ 530, informando que o orçamento foi ultrapassado.
- **Meta de investimento:** A meta de investimento foi definida como R\$ 2000, mas o saldo final foi de R\$ 3473.40, ou seja, a meta foi superada.
- **Resumo das Despesas:** As despesas por categoria são apresentadas com os valores correspondentes.

Conclusão:

Com esse relatório, o usuário pode ter uma visão clara sobre sua situação financeira, sabendo se atingiu as metas de investimento e orçamento, além de ter um resumo detalhado das suas despesas mensais.

Módulo 8

DESENVOLVIMENTO WEB

DESENVOLVIMENTO WEB

DESENVOLVIMENTO WEB



Desenvolvimento web é o processo de criação e manutenção de sites e aplicativos da web. Isso envolve a construção da estrutura, design visual, desenvolvimento de funcionalidades e implementação de recursos interativos. O desenvolvimento web pode ser dividido em duas categorias principais: front-end e back-end.

- **Front-end:** Também conhecido como desenvolvimento do lado do cliente, o front-end é responsável por tudo o que o usuário vê e interage em um site ou aplicativo da web. Isso envolve o uso de tecnologias como HTML, CSS e JavaScript para criar a interface do usuário e fornecer uma experiência visual agradável. O JavaScript é particularmente importante no front-end, pois permite a criação de recursos interativos e dinâmicos.
- **Back-end:** Também conhecido como desenvolvimento do lado do servidor, o back-end lida com a lógica e a funcionalidade por trás de um site ou aplicativo da web. Isso envolve o uso de linguagens de programação como Java, Python, PHP e JavaScript (usando o Node.js) para processar solicitações do cliente, acessar bancos de dados, executar operações de negócios e fornecer os dados necessários para o front-end. O back-end também pode envolver a configuração de servidores e a implementação de medidas de segurança.



O **JavaScript** desempenha um papel fundamental no desenvolvimento web, tanto no front-end quanto no back-end. No front-end, ele é usado para adicionar interatividade, validar formulários, criar animações e manipular elementos da página. No back-end, o JavaScript (usando o Node.js) permite a criação de servidores web, manipulação de solicitações HTTP, acesso a bancos de dados e criação de APIs.

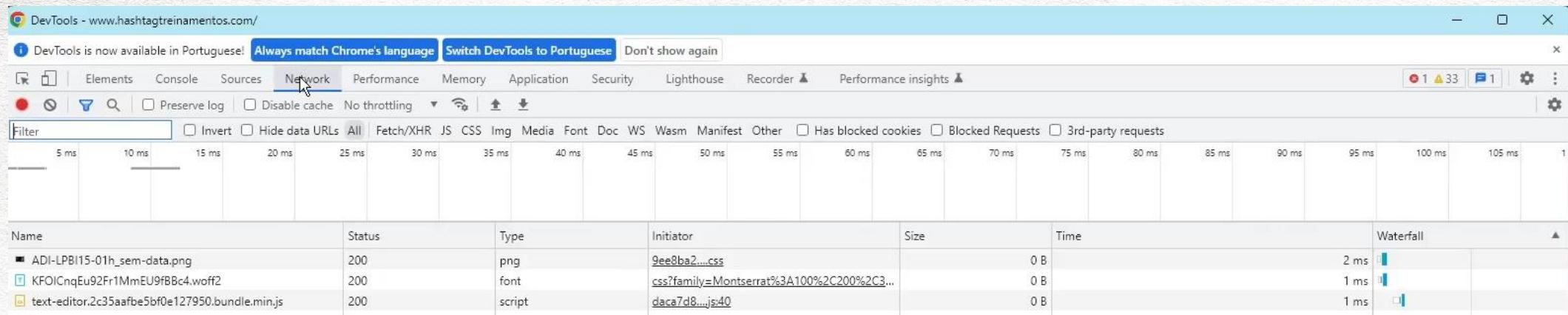
É importante notar que o desenvolvimento web é um campo amplo e em constante evolução, e aprender JavaScript é apenas o primeiro passo. À medida que os iniciantes ganham mais experiência, eles também podem explorar outras tecnologias e frameworks populares, como React, Angular e Vue.js, que facilitam o desenvolvimento web mais avançado e eficiente.



Executando o atalho F12 no seu navegador, ele irá trazer as ferramentas padrões do seu navegador, o DevTools. Nele acessando a opção NETWORK, ele nos retornará o monitoramento do tráfego de rede.

Tráfego de rede

O tráfego de rede refere-se à quantidade de dados que estão se movendo através de uma rede em um determinado momento. Esses dados são transmitidos em pacotes de rede, que são a carga transportada pela rede. O tráfego de rede é um componente essencial para medir, controlar e simular o fluxo de dados em uma rede.

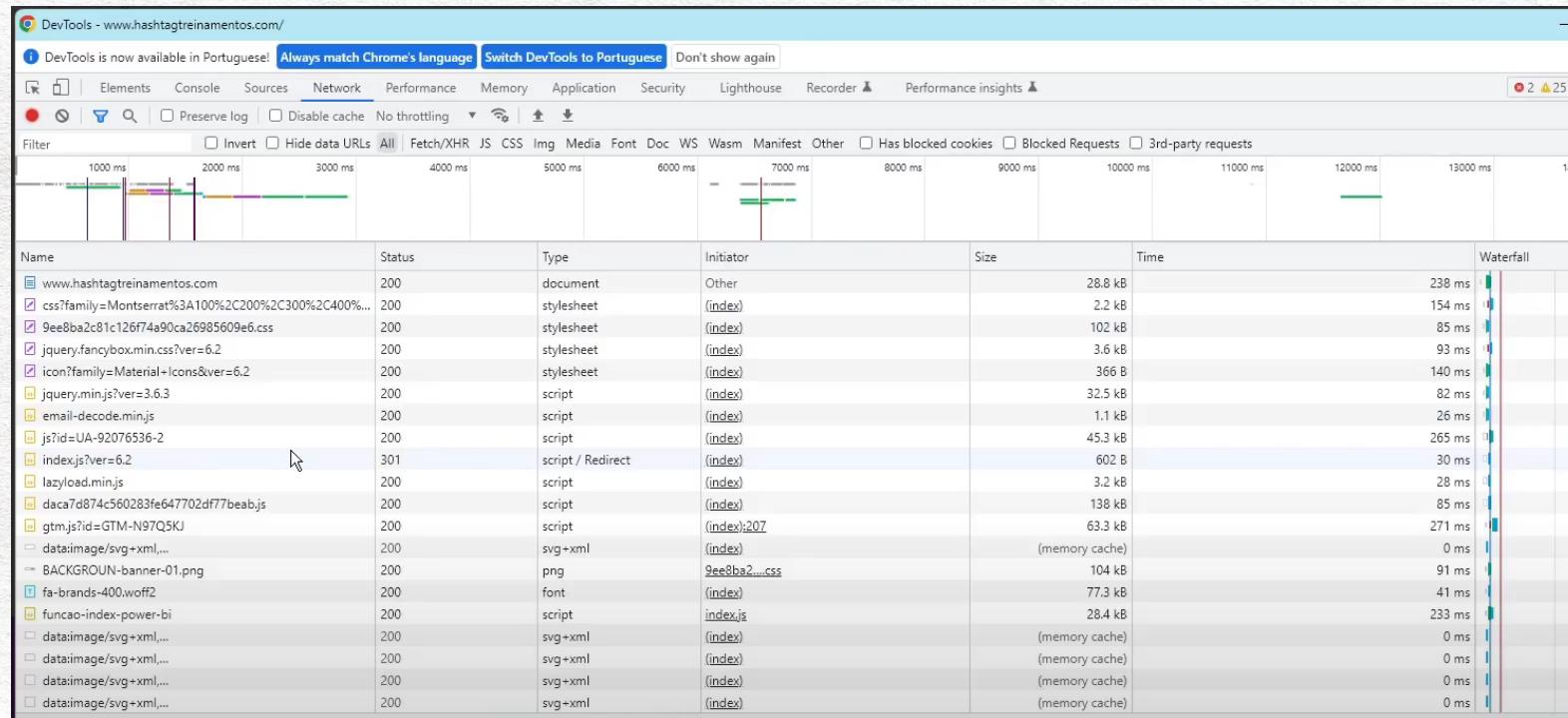


The screenshot shows the Network tab in the Chrome DevTools interface. The tab bar includes Elements, Console, Sources, Network (which is selected), Performance, Memory, Application, Security, Lighthouse, Recorder, and Performance insights. Below the tabs are various controls like Preserve log, Disable cache, and No throttling. A timeline at the top allows for filtering requests by duration. The main table lists network requests with columns for Name, Status, Type, Initiator, Size, Time, and Waterfall. Three requests are listed:

Name	Status	Type	Initiator	Size	Time	Waterfall
ADI-LPB15-01h_sem-data.png	200	png	9ee0ba2....css	0 B	2 ms	[Waterfall]
KFOICnqEu92Fr1MmEU9fBBc4.woff2	200	font	css?family=Montserrat%3A100%2C200%2C3...	0 B	1 ms	[Waterfall]
text-editor.2c35aafbe5bf0e127950.bundle.min.js	200	script	daca7d8....js:40	0 B	1 ms	[Waterfall]

Quando realizamos uma busca, essa ferramenta nos comunica que o computador por meio do meu navegador fez um pedido para um computador na internet que tinha a página que eu queria acessar e ele atendeu esse pedido retornando as informações para o meu navegador.

A internet funciona com base em pedidos e respostas entre Cliente e Servidor.



Requests

Requests (ou requisições) na internet referem-se às solicitações que um dispositivo faz a um servidor para obter informações ou recursos. Quando você digita um endereço de site em seu navegador e pressiona Enter, seu dispositivo envia uma request para o servidor daquele site, solicitando o conteúdo da página. O servidor, por sua vez, processa essa solicitação e envia de volta ao dispositivo a resposta, que é exibida como a página web que você vê.

As requests são essenciais para a comunicação entre dispositivos e servidores na internet. Elas permitem que os dispositivos solicitem dados, arquivos, páginas web, imagens, vídeos e outros recursos que estão armazenados nos servidores. As requests podem ser feitas por meio de diferentes protocolos, como o HTTP (Hypertext Transfer Protocol), que é o protocolo mais comumente usado na web.

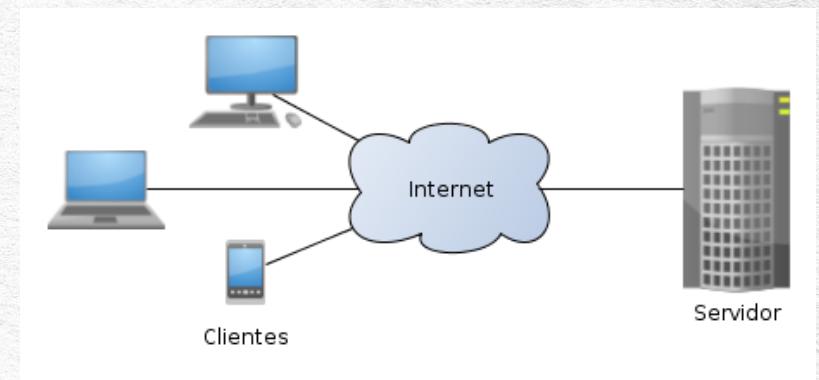
Responses

Responses (ou respostas) na internet referem-se às informações ou recursos que um servidor envia de volta para o dispositivo que fez a request. Quando um dispositivo faz uma request a um servidor, o servidor processa a solicitação e retorna uma response contendo os dados solicitados.

As responses são enviadas pelos servidores como uma forma de fornecer ao dispositivo as informações ou recursos solicitados. Por exemplo, quando você faz uma request para acessar um site, o servidor processa essa solicitação e envia de volta uma response contendo a página web que você deseja visualizar. Essa response pode incluir o código HTML, CSS, JavaScript, imagens e outros recursos necessários para exibir a página corretamente.

Desenvolvimento Web

Consiste no desenvolvimento de aplicações projetadas para funcionar na internet. O exemplo mais imediato porém não exclusivo é o das páginas de internet.



Arquitetura Cliente-Servidor

Arquitetura cliente-servidor é um modelo de comunicação em que um computador (servidor) fornece serviços ou recursos para outros computadores (clientes) que solicitam esses recursos.

Na **arquitetura cliente-servidor**, os servidores são responsáveis por disponibilizar recursos, como arquivos, banco de dados e outros serviços, como servidores de impressão e servidores web. Os clientes, por sua vez, são responsáveis por requisitar esses recursos e serviços aos servidores.

A **comunicação** entre os processos cliente e servidor acontece por meio da troca de mensagens. Geralmente, é utilizado um protocolo de comunicação simples, no qual o cliente envia uma requisição ao servidor, solicitando um determinado serviço. O servidor, então, executa as operações associadas ao serviço e envia uma resposta de volta ao cliente, contendo os dados solicitados ou um código de erro, caso o serviço não possa ser executado.

Arquitetura Cliente-Servidor (continuação)

Essa arquitetura é amplamente utilizada em redes de computadores e sistemas distribuídos. Ela permite a divisão de tarefas e o compartilhamento de recursos, facilitando o acesso a informações e serviços por parte dos clientes. Além disso, a arquitetura cliente-servidor possibilita a escalabilidade, ou seja, a adição de mais servidores para atender a um maior número de clientes, conforme necessário.

A comunicação

TCP/IP

Conjunto de protocolos usados para permitir a comunicação de dados em redes de computadores. Ele é responsável por dividir os dados em pacotes, garantir a entrega confiável dos pacotes encaminhá-los pelo caminho correto na rede usando endereços IP.

O protocolo TCP/IP é composto por dois protocolos principais: o IP (Internet Protocol) e o TCP (Transmission Control Protocol). O IP é responsável por determinar o endereço IP de cada dispositivo conectado à rede, permitindo que eles se comuniquem e troquem dados entre si. Já o TCP é responsável pela entrega dos dados, estabelecendo uma conexão entre a origem e o destino e dividindo os dados em pacotes menores.

O funcionamento do **TCP/IP** se dá através de 4 camadas:

- **Camada de Aplicação:** Nesta camada, os programas conversam entre si usando protocolos como SMTP (para e-mails), FTP (para transferência de arquivos) e HTTP (para navegar na internet). Cada tipo de programa se comunica com um protocolo diferente da camada de Aplicação, dependendo do seu propósito.
- **Camada de Transporte:** Nesta camada, o protocolo da camada de Aplicação se comunica com o protocolo da camada de Transporte, geralmente o TCP. O TCP é responsável por pegar os dados enviados pela camada de Aplicação, dividindo-os em pacotes e enviando-os para a camada inferior, a camada da Internet. Durante o recebimento dos dados, o TCP também é responsável por colocar os pacotes em ordem e verificar a integridade dos dados.
- **Camada da Internet:** Nesta camada, o protocolo IP pega os pacotes recebidos da camada de Transporte e adiciona informações de endereço virtual, como o endereço do computador de origem e o endereço do computador de destino. Esses endereços virtuais são chamados de endereços IP. Em seguida, os pacotes são enviados para a camada inferior, a Interface de Rede.
- **Interface de Rede / Link:** Nesta camada, os pacotes são enviados para a rede física, onde são transmitidos para o destino. Quando os dados chegam à Interface de Rede no destino, eles são chamados de datagramas. O protocolo IP é responsável por enviar os datagramas para a camada de Transporte do destino.



Front-end e back-end são dois termos importantes no desenvolvimento de software.

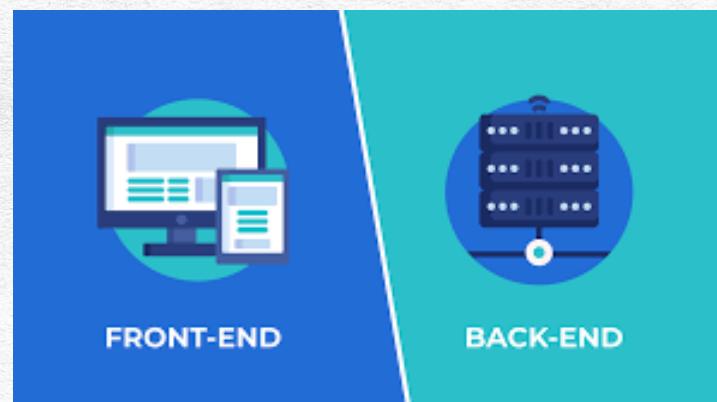
Front-end é a parte visual de um site ou aplicação web, com a qual os usuários interagem diretamente.

O desenvolvedor front-end é responsável por criar a interface gráfica utilizando tecnologias base da web, como HTML, CSS e JavaScript.

O HTML é responsável pela estrutura do conteúdo, o CSS pela aparência e o JavaScript pela interatividade e comportamento do site.

Por exemplo, ao desenvolver um site de e-commerce, o front-end seria responsável por criar a página inicial, a página de produtos, o carrinho de compras, entre outras partes visíveis para o usuário.

O desenvolvedor front-end também pode trabalhar em conjunto com um designer para garantir que a interface seja esteticamente agradável e fácil de usar.



Back-end, por outro lado, é a parte do desenvolvimento que lida com a lógica de negócios, o processamento de dados e a comunicação com o servidor.

O **desenvolvedor back-end** constrói a infraestrutura e os sistemas que suportam o funcionamento do site ou aplicação web. Isso pode incluir o desenvolvimento de bancos de dados, a criação de APIs para comunicação com outras aplicações e a implementação de lógica de autenticação e autorização.

No exemplo do site de e-commerce, o desenvolvedor back-end seria responsável por implementar a funcionalidade de adicionar produtos ao carrinho, processar pagamentos e gerenciar o estoque. Essas tarefas envolvem o uso de linguagens de programação como Java, Python, Ruby ou JavaScript (com o uso de frameworks como Node.js).

É importante ressaltar que **front-end e back-end são áreas complementares** e muitas vezes trabalham em conjunto para criar uma aplicação completa. É possível iniciar com o desenvolvimento front-end e, posteriormente, aprender o desenvolvimento back-end para se tornar um desenvolvedor full-stack, capaz de trabalhar em todas as partes de uma aplicação.

Em resumo, front-end é a parte visual e interativa de uma aplicação web, enquanto back-end lida com a lógica de negócios e a comunicação com o servidor. Ambas as áreas são essenciais para o desenvolvimento de software e existem recursos disponíveis para aprender cada uma delas.

Protocolos de comunicação

HTTP

Protocolo de comunicação mais utilizado na internet, o HTTP (além de sua versão segura, HTTPS) estabelece padrões para comunicação na internet baseadas em "pedidos e respostas" (Requests e Responses).

O **HTTP** depende do TCP/IP para estabelecer e manter a conexão entre o cliente e o servidor, e para garantir que os dados sejam transferidos de forma confiável e sem erros.

O **TCP/IP** é um conjunto de protocolos que é fundamental para o funcionamento da internet. Ele lida com todas as transferências de dados na rede, garantindo que os pacotes de dados sejam enviados corretamente e cheguem ao destino correto.

O **TCP (Transmission Control Protocol)** é responsável pelo controle de transmissão, dividindo os dados em pacotes e garantindo que eles sejam entregues de forma confiável. Já o IP (Internet Protocol) é responsável por transformar esses pacotes em datagramas, incluindo os endereços IP de origem e destino.

TCP/IP e o HTTP se complementam na comunicação de dados na web, sendo que o TCP/IP fornece a base para a transmissão de dados e o HTTP é responsável por definir a estrutura e o formato da comunicação.

O **HTTP** é responsável por estabelecer os pedidos do cliente e as respostas do servidor. Por exemplo, quando você digita o endereço de um site no seu navegador, o navegador envia uma solicitação HTTP para o servidor do site, que retorna a resposta com os dados do site.

O **HTTP** também pode ser usado de forma segura, por meio do HTTPS, que adiciona uma camada de criptografia para proteger a comunicação.

Em resumo, o **HTTP é o protocolo de comunicação mais utilizado na internet** e estabelece padrões para a comunicação na web. Ele depende do TCP/IP para estabelecer e manter a conexão entre o cliente e o servidor.

O **TCP/IP** é responsável pela transmissão confiável dos dados, enquanto o HTTP define a estrutura e o formato da comunicação. Ambos são essenciais para o funcionamento da internet e garantem a transferência segura e confiável dos dados.



FRONT-END e suas tecnologias

Front-end é a parte de um sistema ou site que o usuário interage diretamente, incluindo o design e a funcionalidade da interface.

As principais tecnologias que precisamos conhecer para o desenvolvimento front-end são **HTML, CSS e Javascript**.

Uma linguagem de front-end é uma linguagem de programação utilizada para desenvolver a parte visual e interativa de um site ou aplicação web. Ela é responsável por criar a interface com a qual os usuários interagem diretamente. As linguagens de front-end são fundamentais para definir a estrutura, o layout e o comportamento dos elementos na página.

Começar aprendendo uma linguagem de front-end é uma escolha comum para iniciantes, porque permite ter uma visão mais imediata dos resultados visuais e interativos do desenvolvimento web. Ao aprender uma linguagem de front-end, você estará capacitado para criar páginas web básicas e interativas, o que é um excelente ponto de partida para a sua jornada como desenvolvedor web.



FRONT-END e suas tecnologias

A linguagem **HTML** é um bom ponto de partida para quem quer se tornar um programador front-end.

HTML (HyperText Markup Language) é a linguagem de marcação utilizada para criar a estrutura e o conteúdo de uma página web. Com o HTML, é possível definir elementos como títulos, parágrafos, imagens, links e tabelas.

Aprender HTML permite que você crie e estruture o conteúdo de uma página web de forma organizada e semântica.

Depois de aprender HTML, é recomendado começar a aprender **CSS (Cascading Style Sheets)**. O CSS é uma linguagem de estilo utilizada para controlar a aparência e o layout dos elementos HTML. Com o CSS, você pode definir cores, fontes, tamanhos, margens e posicionamento dos elementos na página. O CSS permite que você torne suas páginas web visualmente atraentes e estilizadas.

Além do HTML e CSS, é importante também aprender JavaScript.

O **JavaScript** é uma linguagem de programação que permite adicionar interatividade e funcionalidade a uma página web. Com o JavaScript, você pode manipular elementos HTML, responder a eventos do usuário, fazer requisições a servidores e muito mais. O JavaScript permite que você crie páginas web dinâmicas e responsivas, proporcionando uma experiência interativa aos usuários.

HTML

HTML (HyperText Markup Language) é a linguagem de marcação utilizada para criar a estrutura e o conteúdo de uma página web. Com o HTML, é possível definir elementos como títulos, parágrafos, imagens, links e tabelas.

Esses elementos são organizados em tags, que são inseridas no código HTML para indicar a função de cada elemento. Por exemplo, a tag `<h1>` é utilizada para definir um título de nível 1, a tag `<p>` é utilizada para criar um parágrafo, e assim por diante.

Além dos elementos básicos, o HTML também permite o uso de atributos. Os atributos HTML são palavras especiais que fornecem informações adicionais sobre os elementos e definem características ou propriedades adicionais desses elementos.

Eles são especificados dentro da tag de abertura do elemento, com um nome e um valor. Por exemplo, o atributo `src` é utilizado na tag `` para indicar o caminho ou URL da imagem a ser exibida no documento.

Outros atributos comuns incluem `width` e `height`, que definem a largura e altura de um elemento, respectivamente. Existem também algumas tags especiais no HTML. A tag `
` é utilizada para quebrar uma linha de texto, sendo mais comum o uso da tag fechada `
`, que é suportada tanto em HTML quanto em XHTML. A tag `` é utilizada para destacar uma parte do texto, fazendo com que ele fique em negrito no navegador web.

HTML

Arquivo index.html

O arquivo index.html é a página padrão dentro de um diretório em servidores de websites. Quando uma pasta é acessada sem especificar o nome de um arquivo específico, o servidor procura pelo arquivo index.html e o entrega para o visitante. A palavra "index" vem do inglês e significa "índice". Nesse contexto, o arquivo index.html é considerado a página principal, que geralmente contém links para todas as outras páginas do site.

Uma tag é representada por **<> tag </>**.

Para montarmos a estrutura básica de um arquivo HTML, podemos digitar **! + ENTER** e o VS Code irá preencher e retornar a estrutura padrão do nosso HTML.



```
① index.html > ② html > ③ head > ④ meta
  1  <!DOCTYPE html>
  2  <html lang="en">
  3  <head>
  4    <meta charset="UTF-8">
  5    <meta http-equiv="X-UA-Compatible" content="IE=edge">
  6    <meta name="viewport" content="width=device-width, initial-scale=1.0">
  7    <title>Document</title>
  8  </head>
  9  <body>
 10    |
 11  </body>
 12 </html>
```

ESTRUTURA DO HTML

Vamos analisar cada parte dessa estrutura que são as tags de metadados:

- **<!DOCTYPE html>**: Essa declaração define o tipo de documento como HTML5. Ela é a primeira linha em um documento HTML e informa aos navegadores qual versão do HTML está sendo utilizada.
- **<html lang="pt-br">**: A tag <html> é o elemento raiz de um documento HTML. Ela engloba todo o conteúdo da página. O atributo lang é utilizado para definir o idioma do documento, nesse caso, "pt-br" indica que o idioma é o português do Brasil.
- **<head>**: A tag <head> é utilizada para conter informações sobre o documento, como o título da página, meta tags, scripts e links para folhas de estilo.
- **<title>Título da página</title>**: A tag <title> é utilizada para definir o título da página, que é exibido na barra de título do navegador e nos resultados de busca. É importante escolher um título descritivo e relevante para a página.
- **<meta charset="utf-8">**: A tag <meta> é utilizada para definir metadados sobre o documento. Nesse caso, o atributo charset define o conjunto de caracteres utilizado na página como UTF-8, que suporta uma ampla gama de caracteres e é recomendado para páginas web.

ESTRUTURA DO HTML

- **<meta name="" content="">**: A tag <meta name> é utilizada para especificar metadados sobre o documento. Ela possui um atributo name que define o tipo de metadado e um atributo content que especifica o valor desse metadado.
- **<meta http-equiv="" content="">**: Já a tag <meta http-equiv> é utilizada para especificar informações equivalentes a cabeçalhos HTTP. Ela possui um atributo http-equiv que define o nome do cabeçalho HTTP e um atributo content que especifica o valor desse cabeçalho.
- **<body>**: A tag <body> é utilizada para definir o conteúdo principal da página, como texto, imagens, links e outros elementos. Todo o conteúdo visível na página deve ser colocado entre as tags de abertura e fechamento do <body>.

Dentro do <body>, você pode adicionar elementos HTML para criar a estrutura e o conteúdo da página, como cabeçalhos <h1>, parágrafos <p>, imagens , links <a>, listas ou , tabelas <table>, entre outros.

A estrutura básica do HTML fornece um esqueleto para a página web, permitindo que você adicione e organize o conteúdo de acordo com a sua necessidade. É importante entender essa estrutura para desenvolver páginas HTML corretamente e garantir uma experiência consistente para os usuários.

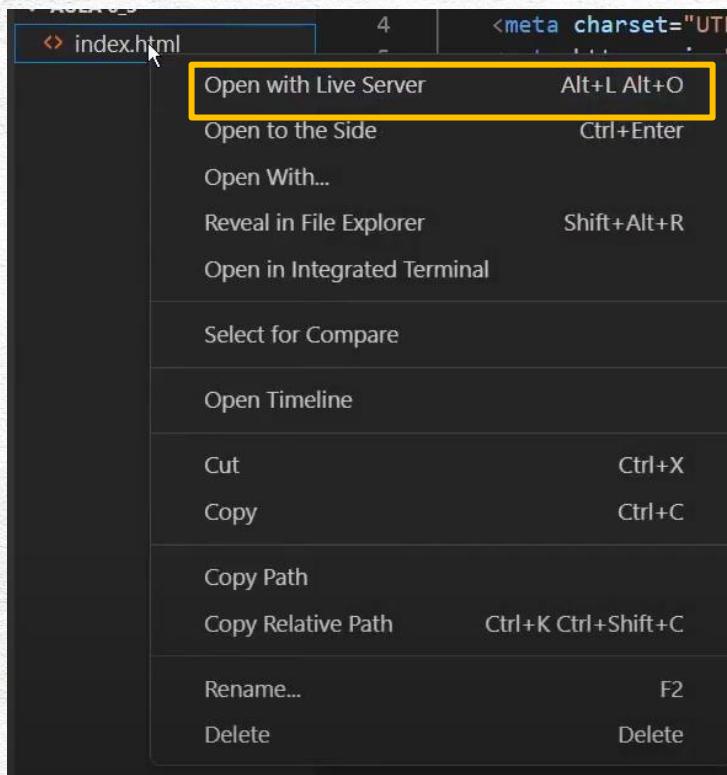
ESTRUTURA BÁSICA:

```
<!DOCTYPE html>
<html>
  <head>
    </head>
  <body>
    </body>
  </html>
```



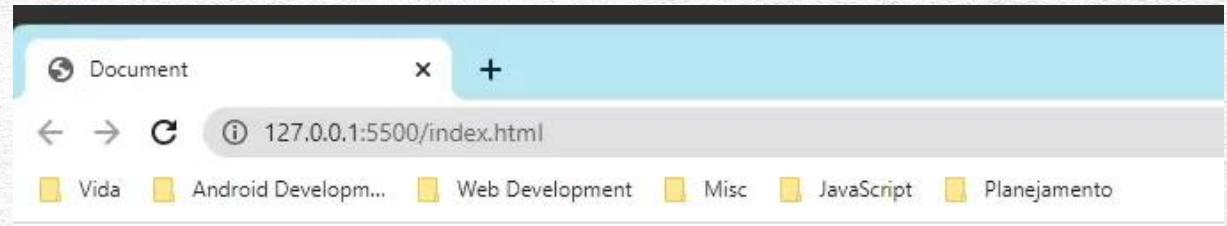
O Vs Code possui uma ferramenta muito interessante para acompanhamos as alterações que estamos realizando no arquivo HTML. Clicando com o botão direito em cima do arquivo index.html irá aparecer a opção **Open with Live Server**, clique nela e ela abrirá uma página no navegador linkado ao seu arquivo.

- Lembrando que você precisará ter instalado a **extensão LIVE SERVER** para conseguir utilizar esse modo.



Dentro da tag body, iremos criar um título utilizando a tag `<h1>` conteúdo `</>` , salvar o nosso arquivo e verificar qual a mudança que houve na nossa página.

```
index.html > ...
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8" />
5      <meta http-equiv="X-UA-Compatible" content="IE=edge" />
6      <meta name="viewport" content="width=device-width, initial-scale=1.0" />
7      <title>Document</title>
8  </head>
9  <body>
10     <h1>Título da minha primeira página!</h1>
11  </body>
12 </html>
```



Título da minha primeira página!

Com esse processo você acaba de fazer a sua primeira página de internet!!

As tags <h1>, <h2>, <h3>, <h4>, <h5> e <h6> são utilizadas no HTML para criar títulos e subtítulos em uma página.

Cada uma dessas tags representa um nível de hierarquia, sendo o <h1> o título principal e o <h6> o subtítulo de menor importância. Aqui está uma explicação mais detalhada sobre o uso dessas tags em português:

- **<h1>**: A tag <h1> é utilizada para definir o título principal da página. Geralmente, é recomendado utilizar apenas um <h1> por página, pois ele representa o título mais importante e define o contexto geral do conteúdo.
- **<h2>**: A tag <h2> é utilizada para definir subtítulos que estão hierarquicamente abaixo do título principal. Ela é usada para dividir o conteúdo em seções menores e mais específicas.
- **<h3>**: A tag <h3> é utilizada para definir subtítulos que estão hierarquicamente abaixo do <h2>. Ela é usada para dividir o conteúdo em subseções dentro de uma seção.
- **<h4>, <h5>, <h6>**: As tags <h4>, <h5> e <h6> são utilizadas para definir subtítulos que estão hierarquicamente abaixo do <h3>. Elas são usadas para dividir o conteúdo em níveis de subseções, sendo o <h4> o subtítulo de nível mais alto e o <h6> o subtítulo de nível mais baixo.

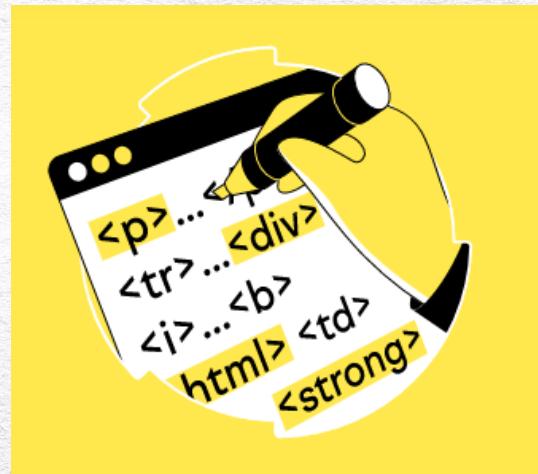
É importante ressaltar que o uso adequado das tags de títulos e subtítulos é fundamental para a acessibilidade e SEO (Search Engine Optimization) de um site. Os títulos e subtítulos ajudam os usuários a entenderem a estrutura do conteúdo e a navearem pelo site de forma mais eficiente. Além disso, os mecanismos de busca utilizam as tags de títulos como um dos fatores para determinar a relevância e o posicionamento das páginas nos resultados de pesquisa.

TAGS

As **tags no HTML** são elementos utilizados para estruturar e organizar o conteúdo de uma página web. Elas são escritas dentro de colchetes angulares (<>) e fornecem instruções ao navegador sobre como exibir e interpretar o conteúdo. As tags são essenciais para criar a estrutura básica de um documento HTML e definir a semântica e o significado do conteúdo.

A importância das tags no HTML está relacionada à forma como elas permitem a organização e estruturação do conteúdo. Quando o navegador encontra uma tag, ele sabe exatamente como interpretar e exibir o conteúdo correspondente na tela. As tags diferenciam um conteúdo HTML de um texto simples, pois elas possuem elementos para estruturar o arquivo e atributos que fornecem informações necessárias a esse elemento.

Além disso, as tags permitem definir a semântica do conteúdo, ou seja, indicar claramente qual é o papel desempenhado por determinado conteúdo na página. Por exemplo, as tags de cabeçalho (como <h1>, <h2>, etc.) são utilizadas para indicar títulos e subtítulos, enquanto as tags de parágrafo (<p>) são utilizadas para agrupar e formatar parágrafos de texto.



O VS Code possui um atalho que quando você escreve o nome da tag, sem utilizar <> ele escreve a tag completa para você colocar apenas as informações que você deseja implementar nela.

TAG IMG

- ****: A tag `` é utilizada no HTML para inserir imagens em um documento. Essa tag é um elemento inline, o que significa que as imagens são organizadas à esquerda uma ao lado da outra.

O **atributo src** especifica o caminho da imagem, enquanto o **atributo alt** fornece um texto alternativo que será exibido caso a imagem não possa ser carregada ou para fins de acessibilidade. O texto alternativo é importante para usuários que utilizam programas leitores de tela, pois permite que eles entendam o conteúdo da imagem mesmo que não possam visualizá-la.

A tag `` não possui uma tag de fechamento (``) porque é considerada um elemento nulo no HTML. Isso significa que ela não pode ter nenhum conteúdo dentro dela, ou seja, ela é autosuficiente.

O **caminho absoluto** e o **caminho relativo** são dois conceitos importantes quando se trata do atributo `src` da tag `` em HTML.

CAMINHO ABSOLUTO E RELATIVO

Um **caminho absoluto** é um caminho completo que especifica o local exato de um arquivo. Ele começa com uma referência à raiz do sistema de arquivos, geralmente representada por uma barra (/) no início do caminho. Por exemplo, um caminho absoluto para uma imagem pode ser algo como /caminho/para/a/imagem.jpg. Esse caminho é independente do local do arquivo HTML em relação ao sistema de arquivos. Portanto, o caminho absoluto sempre leva ao mesmo arquivo, independentemente de onde o arquivo HTML esteja localizado.

Por outro lado, um **caminho relativo** é um caminho que é relativo ao local do arquivo HTML que contém a tag . Ele não começa com uma barra (/) e é baseado na estrutura de diretórios em relação ao arquivo HTML. Existem diferentes maneiras de especificar caminhos relativos, dependendo da localização do arquivo de imagem em relação ao arquivo HTML. Por exemplo:

- Se a imagem estiver no mesmo diretório que o arquivo HTML, você pode simplesmente especificar o nome do arquivo, como imagem.jpg.
- Se a imagem estiver em um diretório acima do arquivo HTML, você pode usar .. para voltar um nível no diretório e, em seguida, especificar o nome do arquivo, como ../imagens/imagem.jpg.
- Se a imagem estiver em um diretório dentro do mesmo diretório pai que o arquivo HTML, você pode especificar o caminho relativo diretamente, como imagens/imagem.jpg.

Ao usar caminhos relativos, é importante levar em consideração a estrutura de diretórios e garantir que o caminho esteja correto em relação ao arquivo HTML. Caso contrário, a imagem não será encontrada e um erro 404 pode ser exibido.

É importante mencionar que o uso de caminhos absolutos ou relativos depende do contexto e dos requisitos do projeto. Caminhos absolutos são úteis quando você precisa especificar um local exato de um arquivo, independentemente de onde o arquivo HTML esteja localizado. Já os caminhos relativos são úteis quando você deseja especificar um local com base na estrutura de diretórios em relação ao arquivo HTML.

TAG P

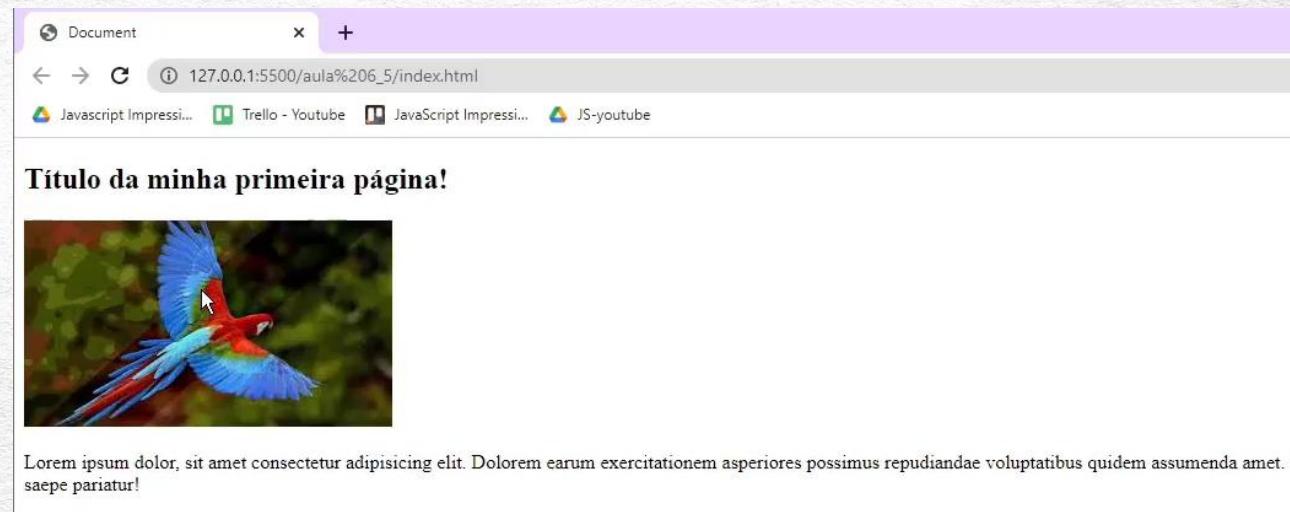
- **<p>**: A tag `<p>` é utilizada no HTML para representar um parágrafo de texto. Ela é um elemento block-level, o que significa que é renderizada como um bloco separado na página, com uma quebra de linha antes e depois do conteúdo do parágrafo. O objetivo principal da tag `<p>` é agrupar e organizar o conteúdo textual em parágrafos.

Um atalho do VS Code é escrever dentro da tag p, **Lorem +Enter** e irá gerar um texto apenas de exemplo para mostrar na sua página.

Após o primeiro contato com o HTML, criamos uma estrutura de página de web da seguinte maneira:

Possuimos um título definido pela tag `<h1>`, uma imagem definida pela tag `` e seus atributos de localização e descrição (`src` e `alt`) e um texto definido pela tag `<p>`. Então essa é a estrutura da nossa página, mas você concorda que ainda é uma página de um texto da cor preta, tela branca e proporções básicas.

```
aula 6_5 > index.html > html > body > h2
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="UTF-8" />
5    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
6    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
7    <title>Document</title>
8  </head>
9  <body>
10 <h2>Título da minha primeira página!</h2>
11 
12 <p>
13   Lorem ipsum dolor, sit amet consectetur adipisicing elit. Dolorem earum
14   exercitationem asperiores possimus repudiandae voluptatibus quidem
15   assumenda amet. Officia aperiam commodi ducimus dignissimos dolores
16   quisquam ea, natus ipsum saepe pariatur!
17 </p>
18 </body>
19 </html>
```



CSS

CSS (Cascading Style Sheets) é uma linguagem de estilo utilizada para definir a aparência e o layout de elementos em uma página web. Ele trabalha em conjunto com o HTML para fornecer estilos e formatação aos elementos do documento. O CSS permite controlar cores, fontes, espaçamento, posicionamento e uma variedade de outros aspectos visuais de um site.

A sintaxe do CSS é baseada em um conjunto de regras. Cada regra é composta por um seletor e um bloco de declaração. O seletor é usado para selecionar os elementos HTML aos quais o estilo será aplicado, enquanto o bloco de declaração contém as propriedades e os valores que definem o estilo desejado.

O **seletor** pode ser um elemento HTML específico, uma classe, um ID ou outros atributos. Além dos seletores de elemento, o CSS também permite o uso de seletores de classe e seletores de ID. As classes e IDs são atributos que podem ser adicionados a elementos HTML para identificá-los e aplicar estilos específicos a eles.



CSS

Existem muitas propriedades CSS disponíveis para estilizar elementos, como **color** (cor do texto), **font-size** (tamanho da fonte), **background-color** (cor de fundo), **margin** (margem), **padding** (preenchimento), entre outras. Cada propriedade possui um valor específico que determina como o estilo será aplicado.

O CSS permite a criação de estilos personalizados para cada elemento HTML em uma página, oferecendo uma grande flexibilidade para a aparência e o layout. Ele também suporta conceitos como herança, especificidade e cascata, que ajudam a controlar como os estilos são aplicados e priorizados.

Em resumo, o CSS é uma linguagem de estilo utilizada para definir a aparência e o layout de elementos em uma página web. Ele trabalha em conjunto com o HTML e permite controlar cores, fontes, espaçamento e outros aspectos visuais. O CSS utiliza uma sintaxe baseada em regras, com seletores e blocos de declaração, e oferece uma ampla variedade de propriedades para estilizar elementos de forma personalizada.

Escrevemos o código CSS e o unimos ao nosso projeto. Assim, os elementos apropriados receberão as mudanças estéticas definidas por essa nova linguagem. Com o CSS, conseguimos tirar a nossa página do tradicional visual do HTML puro (que é muito pouco atraente), e levá-la a patamares impressionantes e únicos!



```
body {  
    font: x-small  
    background: #  
    color: black;  
    margin: 0;  
    padding: 0;
```



TAG STYLE

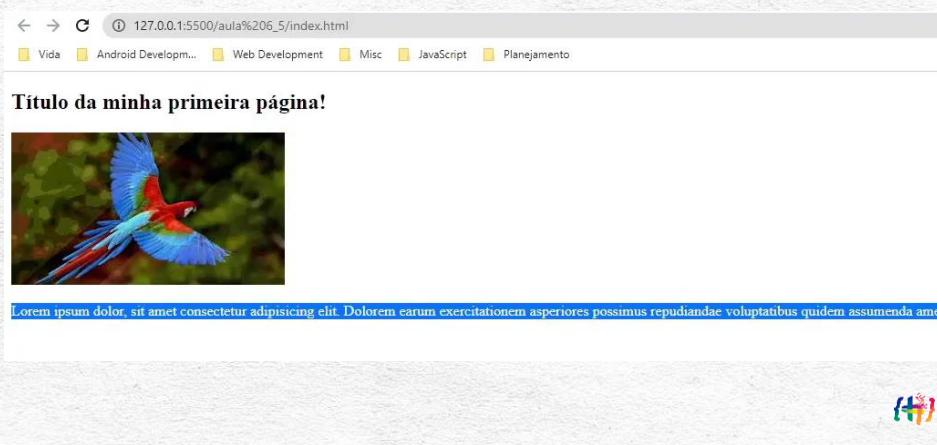
- **<style>**: A tag <style> é utilizada no HTML para incluir estilos CSS diretamente no documento HTML. Ela permite definir regras de estilo que serão aplicadas aos elementos do documento, sem a necessidade de um arquivo externo de estilo.

A tag <style> deve ser colocada dentro da seção <head> do documento HTML, entre as tags de abertura <style> e de fechamento </style>. Dentro do bloco de código da tag <style>, podemos escrever as regras de estilo CSS que desejamos aplicar aos elementos da página.

Dentro da tag <style>, podemos escrever qualquer código CSS válido, incluindo seletores, propriedades e valores. Podemos definir estilos para diferentes elementos HTML, classes, IDs e outros seletores.

É importante lembrar que a tag <style> é mais adequada para estilos simples e pequenos. Para projetos maiores e mais complexos, é recomendado o uso de um arquivo externo de estilo CSS.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <style>
      p {
        color: white;
        background-color: #3282f6;
      }
    </style>
  </head>
  <body>
    <h2>Título da minha primeira página!</h2>
    
    <p>
      Lorem ipsum dolor, sit amet consectetur adipisicing elit. Dolorem earum
      exercitationem asperiores possimus repudiandae voluptatibus quidem assumenda amet
      officia aperiam commodi ducimus dignissimos dolores
      quisquam ea, natus ipsum saepe pariatur!
    </p>
  </body>
</html>
```



Propriedades

Propriedades de CSS são recursos utilizados para definir estilos e características específicas de elementos HTML em uma página web. Elas são responsáveis por controlar a aparência, o layout e o comportamento dos elementos, permitindo personalizar a experiência visual do usuário.

Existem diversas propriedades CSS disponíveis, cada uma com sua função e sintaxe específica. Algumas das propriedades mais comuns incluem:

- **color:** define a cor do texto.
- **font-size:** define o tamanho da fonte.
- **background-color:** define a cor de fundo de um elemento.
- **margin:** define as margens externas de um elemento.
- **padding:** define o espaço interno entre o conteúdo de um elemento e suas bordas.
- **border:** define as bordas de um elemento.
- **width e height:** definem a largura e altura de um elemento.
- **display:** define como um elemento é exibido na página.
- **position:** define o posicionamento de um elemento na página.
- **float:** define o alinhamento horizontal de um elemento em relação a outros elementos.
- **text-align:** define o alinhamento horizontal do texto dentro de um elemento.

Essas são apenas algumas das muitas propriedades disponíveis no CSS. Cada propriedade possui um valor associado, que pode ser uma cor, um tamanho, uma palavra-chave ou uma função específica.

As propriedades CSS podem ser aplicadas a elementos HTML de diferentes maneiras. Podem ser definidas inline, diretamente no atributo style de um elemento, ou podem ser definidas em um arquivo CSS externo e referenciadas no documento HTML.

Propriedade MARGIN

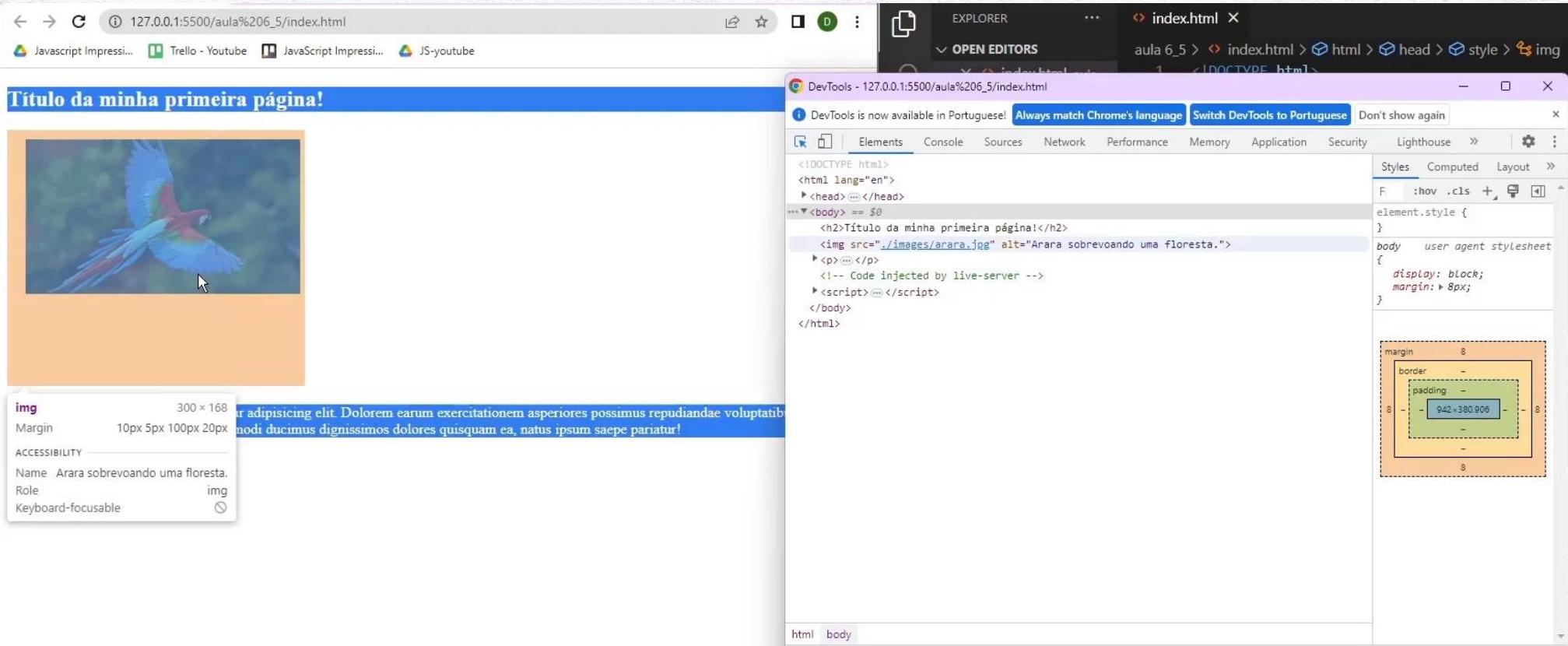
A **propriedade margin** do CSS é utilizada para definir o espaçamento externo de um elemento, ou seja, o espaço ao redor da borda do elemento. Ela permite adicionar margens nas quatro direções: superior (**margin-top**), inferior (**margin-bottom**), esquerda (**margin-left**) e direita (**margin-right**).

A principal diferença entre as propriedades margin-top, margin-bottom, margin-left e margin-right é a direção em que a margem é aplicada. Por exemplo, a propriedade margin-top define o espaçamento na parte superior do elemento, margin-bottom define o espaçamento na parte inferior, margin-left define o espaçamento na parte esquerda e margin-right define o espaçamento na parte direita.

É importante notar que a propriedade margin pode receber diferentes valores, como:

- **Valores absolutos:** podem ser definidos em pixels (px), centímetros (cm), milímetros (mm), polegadas (in), entre outros. Por exemplo: margin-top: 20px;
- **Valores relativos:** podem ser definidos em porcentagem (%) em relação ao elemento pai. Por exemplo: margin-bottom: 50%;
- **Valores automáticos:** o navegador calcula o valor da margem automaticamente. Por exemplo: margin-left: auto;

Através do DevTools (F12) podemos entender o que acontece com o nosso elemento ao utilizarmos a propriedade margin.



O código da nossa página da web está no seguinte formato:

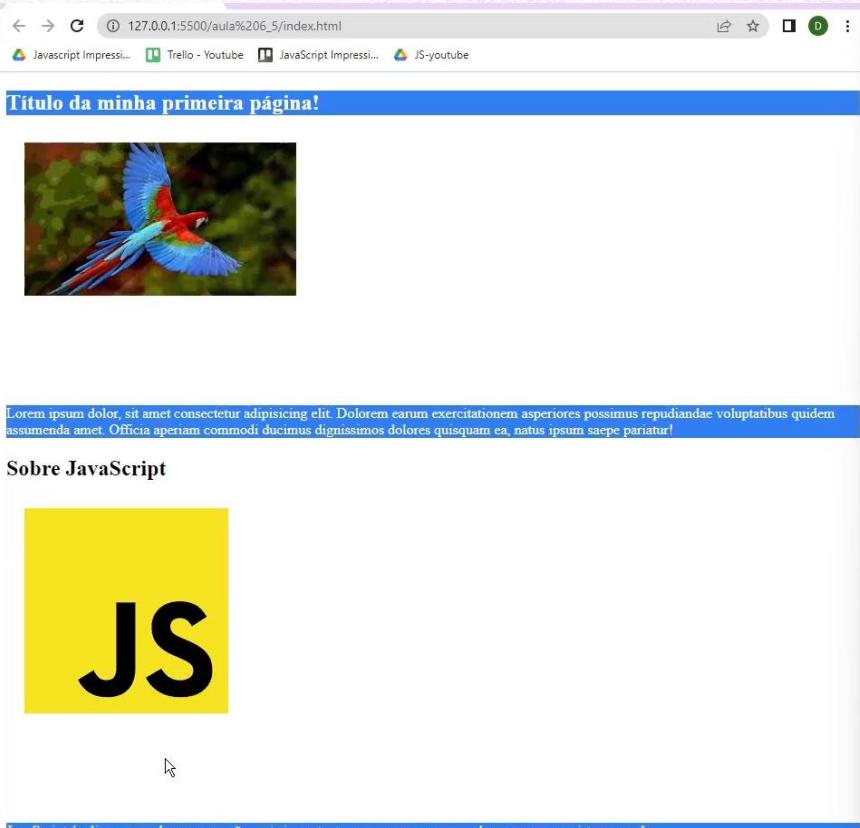
```
5 > index.html > html > body > p
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, ini...
  <title>Document</title>
  <style>
    h2 {
      color: white;
      background-color: #3282f6;
    }
    img {
      margin: 10px 5px 100px 20px;
    }
    p {
      color: white;
      background-color: #3282f6;
    }
  </style>
</head>
<body>
  <h2>Título da minha primeira página!</h2>
  
    Lorem ipsum dolor, sit amet consectetur adipisicing exercitationem asperiores possimus repudiandae voluptu...
    assumenda amet. Officia aperiam commodi ducimus dignissimos. Quia quod est ut labore et dolore magna aliqua...
    quisquam ea, natus ipsum saepe pariatur!
  </p>
</body>
```

A partir desse código iremos adicionar novos elementos a nossa página, como o elemento de tag section.

- **<section>:** A tag <section> do HTML é utilizada para definir seções de documentos, como capítulos, cabeçalhos, rodapés ou qualquer outra seção.
- Ela divide o conteúdo em seções e subseções. A tag <section> é usada quando há a necessidade de dois cabeçalhos, rodapés ou qualquer outra seção no documento. Ela agrupa blocos genéricos de conteúdo relacionado.

A principal vantagem da tag <section> é que ela é um elemento semântico, o que significa que descreve seu significado tanto para o navegador quanto para o desenvolvedor.

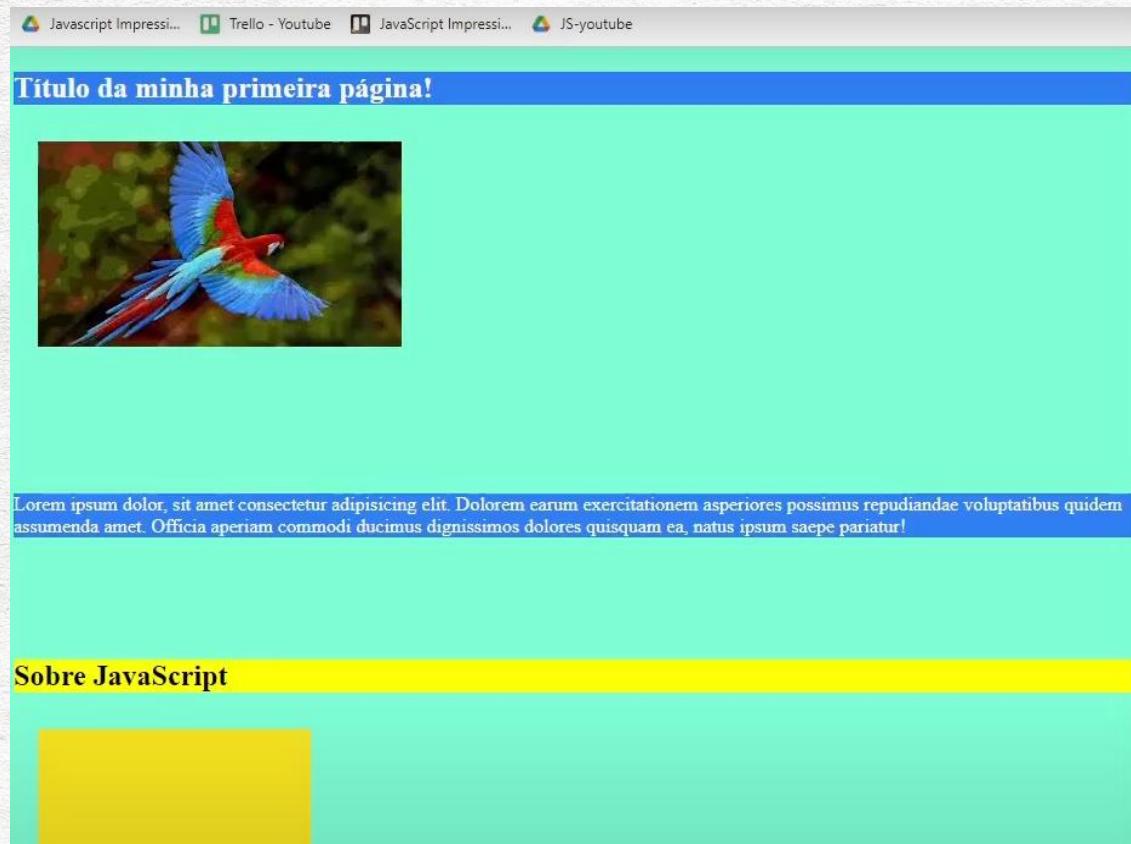
A abordagem da nossa folha de estilo do CSS, irá estilizar da mesma forma os seletores que estamos utilizando de forma genérica, e isso pode começar a ocorrer alguns problemas, pois nem sempre queremos que os elementos possuam as mesmas características.



The screenshot shows a web browser window displaying a local file at 127.0.0.1:5500/aula%206_5/index.html. The page content includes a title "Título da minha primeira página!", a parrot image, a paragraph of placeholder text, and a section titled "Sobre JavaScript" with a JS logo. On the right side, the Visual Studio Code interface is visible, showing the Explorer, Open Editors, and Outline panes. The index.html file is open in the editor, displaying CSS code that styles h2 and p elements with white text and a dark background color (#32829f). The code also includes a meta viewport tag and a section about JavaScript.

```
index.html
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
    <title>Document</title>
    <style>
      h2 {
        color: white;
        background-color: #32829f;
      }
      img {
        margin: 10px 5px 100px 20px;
      }
      p {
        color: white;
        background-color: #32829f;
      }
    </style>
  </head>
  <body>
    <h2>Título da minha primeira página!</h2>
    
    <p>Lorem ipsum dolor, sit amet consectetur adipisicing elit. Dolorem earum exercitationem asperiores possimus repudiandae voluptatibus quidem assumenda amet. Officia apertam commodi ducimus dignissimos dolores quisquam ea, natus ipsum saepe pariatur!</p>
    <section>
      <h3>Sobre JavaScript</h3>
      <img alt="JavaScript logo: JS" data-bbox="185 680 265 820"/>
      <p>JavaScript é a linguagem de programação mais importante para quem quer aprender a programar sistemas web.</p>
    </section>
  </body>
</html>
```

Outra situação que iremos encontrar com a abordagem que estamos aplicando é que irá chegar um momento no qual a nossa tag style será muito longa, pois ela aplicará estilos em diversas tags, e utilizar diversas propriedades.



The screenshot shows a web browser window with the title "Titúlo da minha primeira página!". Inside the page, there is a large image of a colorful parrot in flight. Below the image, there is a yellow horizontal bar with the text "Sobre JavaScript". The browser's address bar shows several tabs, including "Javascript Impressi...", "Trello - Youtube", "JavaScript Impressi...", and "JS-youtube".

On the right side of the image, there is a code editor window titled "aula 6_5 > index.html > html > head > style > body". The code editor displays the following HTML and CSS:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<meta http-equiv="X-UA-Compatible" content="IE=edge" />
<meta name="viewport" content="width=device-width, initial-scale=1.0" />
<title>Document</title>
<style>
h2 {
    color: white;
    background-color: #3282f6;
}
img {
    margin: 10px 5px 100px 20px;
}
p {
    color: white;
    background-color: #3282f6;
}
h1 {
    margin-top: 100px;
    background-color: yellow;
}
body {
    background-color: aquamarine;
}
</style>
</head>
```

Temos dois problemas acontecendo, o primeiro é que a tag style pode aumentar exponencialmente , fazendo com que o nosso arquivo HTML perca a sua funcionalidade de guardar o nosso HTML, ou seja, ele será preenchido por uma linguagem que não é HTML.

O segundo problema é que as definições de estilo serão genéricas para todos os elementos que utilizam o mesmo seletor, e nem sempre queremos aplicar as mesmas estilizações para elementos iguais só que com conteúdo diferentes, ou seja, nem todo parágrafo tem estar com a cor branca e cor de fundo azul.

Para conseguir corrigir os dois problemas que a nossa abordagem gerou iremos alterar o nosso código. Primeiramente iremos separar as responsabilidades do HTML e CSS, criando um arquivo style.css que irá armazenar todo o conteúdo de CSS da nossa página.

Arquivo style.css

O **arquivo style.css** é um arquivo de folha de estilos em cascata (CSS) utilizado para definir a aparência e o estilo de um documento HTML ou XML. O CSS é uma linguagem de estilo que descreve como os elementos de um documento devem ser apresentados na tela. Ele permite controlar a cor, o layout, a fonte, o tamanho, o espaçamento e outros aspectos visuais dos elementos.

A estrutura básica de um arquivo style.css é composta por um conjunto de regras de estilo. Cada regra é composta por um seletor e um bloco de declarações. O seletor identifica o(s) elemento(s) HTML/XML que serão estilizados e o bloco de declarações contém as propriedades e os valores que serão aplicados a esses elementos.

Arquivo style.css

O arquivo style.css é vinculado ao documento HTML/XML utilizando a tag **<link>** no cabeçalho do documento. O **atributo href** especifica o caminho para o arquivo CSS. Dessa forma, o arquivo style.css será aplicado ao documento HTML/XML, definindo o estilo dos elementos conforme as regras especificadas.

A **propriedade rel** do elemento **<link>** é utilizada para especificar a relação entre o documento atual e o documento/link referenciado. Ela define o tipo de relação que o link possui com o documento/link referenciado. A propriedade rel é um atributo obrigatório no elemento **<link>**.

```
aula 6_5 > index.html > ...
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8" />
5      <meta http-equiv="X-UA-Compatible" content="IE=edge" />
6      <meta name="viewport" content="width=device-width, initial-scale=1.0" />
7      <title>Document</title>
8      <link rel="stylesheet" href=".//style.css">
9  </head>
10 <body>
11     <h2>Título da minha primeira página!</h2>
12     
13     <p>
14         Lorem ipsum dolor, sit amet consectetur adipisicing elit. Exercitationem asperiores possimus repudiandae voluptatem assumenda amet. Officia aperiam commodi ducimus dignissimos quisquam ea, natus ipsum saepe pariatur!
15     </p>
16     <section>
17         <h3>Sobre JavaScript</h3>
18         
19         <p>
20             JavaScript é a linguagem de programação mais importante para aprender a programar sistemas web
21         </p>
22     </section>
23 </body>
24 </html>
```

Agora vamos corrigir o segundo problema que a nossa abordagem estava gerando, que é estilizar de maneiras diferentes elementos iguais. Vamos fazer isso com os parágrafos da nosso HTML.

Inline Styling: O Inline Styling é uma forma de aplicar estilos diretamente a um elemento HTML, utilizando o atributo style. Com o Inline Styling, é possível definir as propriedades de estilo de um elemento diretamente no próprio elemento, sem a necessidade de um arquivo de estilo separado.

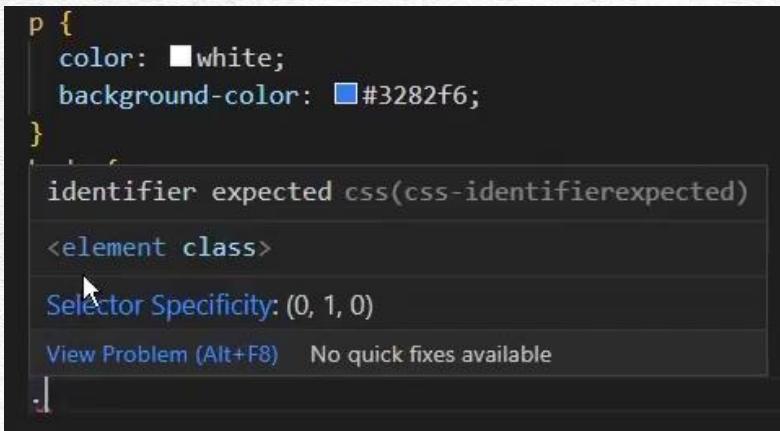
```
<p style="color: red; font-size: 16px;">Este é um parágrafo com estilo inline.</p>
```

Porém isso não é o recomendado, já que estamos separando as responsabilidades do HTML e CSS.

Seletores: Os seletores do CSS são utilizados para selecionar os elementos HTML aos quais se deseja aplicar um estilo. Existem vários tipos de seletores, como seletores de tipo, seletores de classe, seletores de ID, seletores de atributo, entre outros. Cada seletor tem uma sintaxe específica e permite selecionar elementos com base em diferentes critérios.

Classes: As classes do CSS são utilizadas para agrupar elementos que possuem características semelhantes e aplicar estilos a esses elementos de forma conjunta. Uma classe é definida utilizando o seletor de classe, que é representado pelo caractere (.), seguido pelo nome da classe.

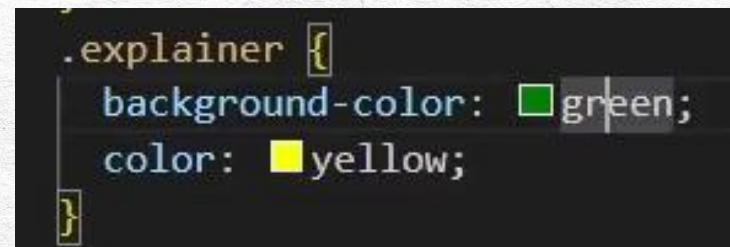
```
<p class="destaque">Este é um parágrafo com a classe destaque.</p>
<p class="destaque">Este é outro parágrafo com a classe destaque.</p>
```



A screenshot of a code editor showing a CSS selector error. The code is as follows:

```
p {
  color: white;
  background-color: #3282f6;
}
```

The cursor is positioned after the closing brace of the first rule. A tooltip message is displayed: "identifier expected css(css-identifierexpected)". Below the code, there is a status bar with the text "Selector Specificity: (0, 1, 0)" and "View Problem (Alt+F8) No quick fixes available".



A screenshot of a code editor showing a CSS class definition. The code is as follows:

```
.explainer {
  background-color: green;
  color: yellow;
}
```

The cursor is positioned after the closing brace of the class definition. The background of the code editor has a gradient from dark blue to light blue.

Identificadores: Os identificadores do CSS são utilizados para selecionar elementos HTML com base em um ID único. Um identificador é definido utilizando o seletor de ID, que é representado pelo caractere (#), seguido pelo nome do ID. Diferente das classes, um ID deve ser único dentro do documento HTML.

```
<h1 id="titulo">Título da página</h1>
```

Nesse exemplo, o elemento `<h1>` possui o ID `titulo`, o que permite selecioná-lo de forma exclusiva utilizando o seletor de ID.

```
<element id="page-title">  
Selector Specificity: (1, 0, 0)  
#page-title
```

```
#page-title {  
    background-color: brown;  
}
```

Ao desenvolver uma página da web, é fundamental entender a importância da estruturação do conteúdo para uma melhor organização e acessibilidade. Uma das maneiras mais eficazes de alcançar isso é através do uso de tags de cabeçalho HTML. As tags de cabeçalho não apenas definem a hierarquia do conteúdo, mas também desempenham um papel crucial na otimização para mecanismos de busca.

Vamos explorar as diferentes tags de cabeçalho HTML, desde o **<h1>** até o **<h6>**, e entender como elas são utilizadas para segmentar e organizar o conteúdo em uma página da web. Ao compreender o propósito e a aplicação de cada uma dessas tags, você poderá criar páginas da web mais estruturadas e acessíveis, beneficiando tanto os usuários quanto os motores de busca.

Essas são tags HTML de cabeçalho que são usadas para definir a estrutura e hierarquia do conteúdo em uma página da web. Aqui está uma explicação sobre cada uma delas:

<h1>: É a tag de cabeçalho mais importante e define o título principal da página. Normalmente é usado apenas uma vez em uma página para indicar o título principal.

<h2>: É uma tag de cabeçalho de segundo nível e é usada para subtítulos importantes. Geralmente é usado para seções importantes do conteúdo, que são secundárias ao título principal.

<h3>: É uma tag de cabeçalho de terceiro nível, usada para subdivisões de seções em uma página. É útil para organizar o conteúdo em seções menores e mais específicas.

<h4>: É uma tag de cabeçalho de quarto nível. É usada para subseções de conteúdo que estão dentro de seções h3, h2 ou h1.

<h5>: É uma tag de cabeçalho de quinto nível, que é menos comum de ser usada. Normalmente, você não a encontra com frequência, mas é utilizada para subdivisões ainda mais específicas do conteúdo.

<h6>: É a tag de cabeçalho de sexto nível, usada para títulos ou subtítulos menos importantes ou para especificações muito específicas dentro do conteúdo.

Essas tags não apenas ajudam a estruturar e organizar o conteúdo de uma página da web, mas também têm implicações em termos de SEO (Search Engine Optimization), pois os mecanismos de busca geralmente dão mais peso ao conteúdo dentro de tags de cabeçalho, especialmente **<h1>**, em comparação com o texto normal na página.

```
<h1>Título na minha página</h1>
<h2>subtítulo na minha página</h2>
<h3>Tag h3 e seu texto</h3>
<h4>Texto tag h4</h4>
<h5>Tag h5</h5>
<h6>Tag h6 finalmente</h6>
```

Título na minha página
subtítulo na minha página
Tag h3 e seu texto
Texto tag h4
Tag h5
Tag h6 finalmente

A tag **<p>** é usada para criar parágrafos de texto em HTML. Além de ser usada para criar parágrafos de texto, a tag **<p>** é uma das tags mais comuns e básicas em HTML e é fundamental para a estruturação e organização do conteúdo textual em uma página da web.

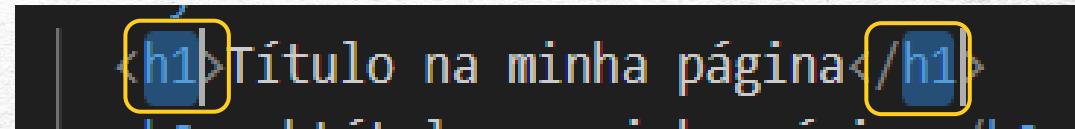
```
<p>Texto da minha página</p>
<p>Segundo texto da minha página</p>
```

Texto da minha página
Segundo texto da minha página



Edições rápidas

O atalho CTRL+D seleciona a próxima ocorrência da palavra ou sequência de caracteres que está atualmente selecionada no editor de texto, facilitando a edição em várias partes do documento.



A tag **<style>** em HTML é usada para definir estilos específicos para um elemento HTML. No exemplo abaixo, a propriedade **font-weight: bold** dentro do atributo **style** da primeira tag **<p>** indica que o texto contido dentro dessa tag será exibido em negrito. Já o segundo **<p>** não possui estilos definidos, portanto seu texto será exibido com o estilo padrão do navegador.

```
<p style="font-weight: bold">Texto da minha página</p>
<p>Segundo texto da minha página</p>
```

Texto da minha página
Segundo texto da minha página

A tag **<div>** em HTML é usada como um contêiner genérico para agrupar e estruturar elementos de uma página da web. Ela não possui significado semântico próprio, mas é frequentemente utilizada em conjunto com o atributo **class** ou **id** para aplicar estilos CSS ou para facilitar a manipulação do conteúdo por meio de JavaScript. A **<div>** é essencial para organizar o layout e a estrutura de uma página da web, permitindo a criação de seções distintas e facilitando a manutenção do código.

```
<div style="margin: 10px">Terceiro texto da minha página</div>
<div>Quarto texto da minha página</div>
```

Terceiro texto da minha página
Quarto texto da minha página

O estilo **<div style="margin: 10px">** define uma margem de 10 pixels ao redor do elemento **<div>**. Isso significa que haverá um espaço de 10 pixels entre o conteúdo dentro do **<div>** e as bordas do próprio **<div>**, proporcionando um espaço visualmente agradável ao redor do elemento.

O uso excessivo de elementos `<div>` pode tornar o código HTML menos semântico e dificultar a manutenção e compreensão do código. Aqui estão algumas razões pelas quais devemos tomar cuidado ao utilizar muitos elementos `<div>`:

Semântica comprometida: A tag `<div>` não tem significado semântico próprio. Usá-la em excesso pode fazer com que o código perca sua semântica, tornando-o menos acessível para leitores assistivos e motores de busca.

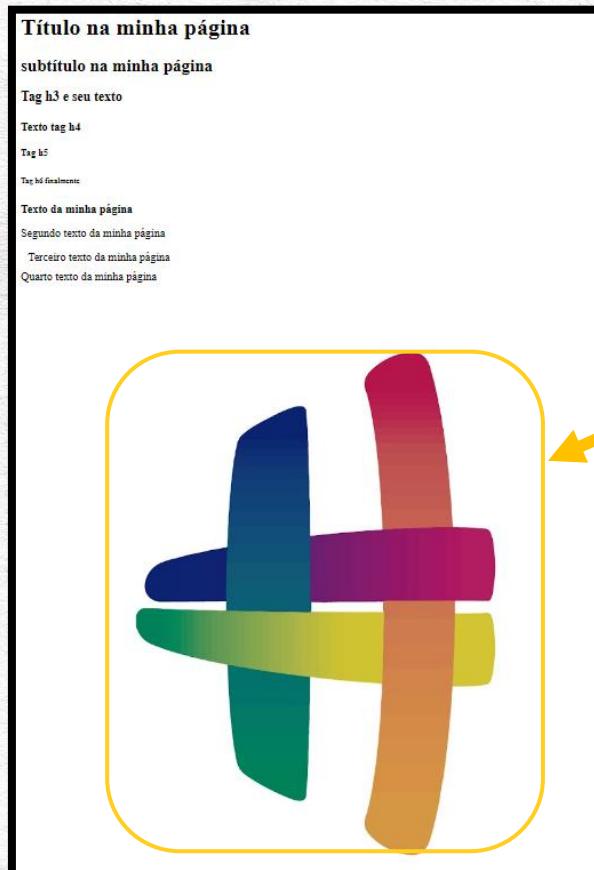
Dificuldade de manutenção: Com muitos elementos `<div>`, o código pode se tornar confuso e difícil de entender. Isso pode aumentar a probabilidade de erros durante a manutenção e dificultar a colaboração entre membros da equipe.

Desempenho prejudicado: Um grande número de elementos `<div>` pode resultar em código HTML mais pesado, o que pode afetar o desempenho do site, especialmente em dispositivos com recursos limitados, como smartphones e tablets.

Flexibilidade limitada: O uso excessivo de `<div>` pode resultar em estruturas de layout rígidas e difíceis de adaptar para diferentes tamanhos de tela e dispositivos. Isso pode dificultar a criação de sites responsivos e compatíveis com dispositivos móveis.

Em vez de depender exclusivamente de `<div>`, é recomendável utilizar elementos HTML semânticos apropriados, como `<header>`, `<nav>`, `<main>`, `<article>`, `<section>`, entre outros, sempre que possível. Isso não apenas melhora a semântica do código, mas também facilita a manutenção e a compreensão do layout da página. Além disso, o uso criterioso de `<div>` em conjunto com classes e IDs significativos pode ser benéfico para aplicar estilos CSS específicos e para manipular o conteúdo com JavaScript.

A tag **** é usada em HTML para incorporar imagens em uma página da web. No exemplo fornecido, o atributo **src** especifica o URL da imagem que será exibida, enquanto o atributo **alt** fornece um texto alternativo que é exibido caso a imagem não possa ser carregada ou para usuários com deficiência visual, permitindo-lhes entender o conteúdo da imagem. No exemplo, o atributo **alt** está vazio, o que significa que não há texto alternativo específico fornecido para essa imagem.



```

```

Nessa aula você utilizará o código abaixo, ele está disponível para download na aula da plataforma.

```
<body>
  <button>Botão de Exemplo</button>
  <div id="div-conteudo-1">Conteúdo 1</div>
  <div class="classe-grupo-amarelo">
    Conteúdo 2
    <div>Conteúdo 3</div>
  </div>
  <div id="div-conteudo-4-7">
    <div>Conteúdo 4</div>
    <span style="background-color: blue"> Conteúdo 5 </span>
    <span> Conteúdo 5.5 </span>
    <div class="classe-texto-azul">Conteúdo 6</div>
    <div>Conteúdo 7</div>
  </div>
  <div>
    <nav>
      <ul>
        <li>Item 1</li>
        <li>Item 2</li>
        <li>Item 3</li>
        <div class="classe-grupo-amarelo classe-texto-azul">
          <li>Item 4</li>
        </div>
      </ul>
    </nav>
  </div>
</body>
```

```
1   body { background-color: aquamarine; }
2
3   #div-conteudo-1 { background-color: red; }
4
5   .classe-grupo-amarelo { background-color: yellow; }
6
7   /* div {
8     background-color: pink;
9   } */
10
11  .classe-texto-azul { color: blue; }
12
13  .classe-grupo-amarelo.classe-texto-azul { color: green; }
14
15  #div-conteudo-4-7 div { background-color: blueviolet; }
16
17  #div-conteudo-4-7 > div { color: orange; }
18
19  button {
20    background-color: cornflowerblue;
21    padding: 0;
22    margin-bottom: 20px;
23    border: 10px solid red;
24  }
25
26  button:hover { background-color: aqua; }
```



Esse código HTML e CSS é usado para ilustrar e ensinar sobre os conceitos de elementos em nível de bloco e elementos em linha.

HTML:

- Define a estrutura do documento, incluindo tags como **<div>**, **<button>**, **<nav>**, ****, e ****.
- Além disso, vincula um arquivo de estilo externo chamado "style.css" para estilizar o conteúdo.

CSS:

- Define estilos visuais para os elementos HTML.
- Estilos como cores de fundo (**background-color**), cor do texto (**color**), margens (**margin**), preenchimento (**padding**), e bordas (**border**) são aplicados.
- Seletores específicos como IDs (#) e classes (.) são usados para segmentar diferentes elementos HTML.

Aqui está um resumo de alguns pontos importantes:

- A cor de fundo do **<body>** é definida como "aquamarine".
- Diferentes elementos **<div>** têm cores de fundo específicas.
- Um botão é estilizado com uma cor de fundo específica, preenchimento zero, e uma borda vermelha.
- Há uma regra para alterar a cor de fundo do botão quando ele é "hovered" (passagem do mouse sobre ele).
- Algumas regras de estilo são aplicadas a elementos dentro de determinadas classes e IDs, como **#div-conteudo-4-7 div** e **.classe-grupo-amarelo.classe-texto-azul**.
- A estrutura HTML contém uma mistura de elementos de bloco (como **<div>**, **<nav>**, ****) e elementos de linha (como **** dentro de um **<div>**).

Esse código serve para demonstrar como os diferentes tipos de elementos HTML podem ser estilizados visualmente usando CSS, e como o posicionamento e o comportamento dos elementos podem ser manipulados com base em suas propriedades de bloco ou linha.

A tag **** é uma tag HTML que é usada para agrupar elementos de texto em linha sem semanticamente alterar ou adicionar significado ao conteúdo. Ela é usada principalmente para aplicar estilos ou comportamentos específicos a partes de um texto dentro de um parágrafo, uma linha ou um bloco de texto.

Os elementos **<div>** e **** são elementos HTML comuns usados para organizar e estruturar o conteúdo de uma página da web. Ambos são elementos de nível de bloco e de linha, dependendo de como são usados em relação ao contexto do documento HTML. Aqui está uma explicação sobre ambos:

Elementos de Nível de Bloco:

<div>: O elemento **<div>** é um elemento de nível de bloco, o que significa que ele ocupa toda a largura disponível do contêiner pai e inicia em uma nova linha. Ele é frequentemente usado para agrupar e estruturar blocos de conteúdo, como seções de uma página da web. Exemplos incluem cabeçalhos, parágrafos, listas e outros elementos de bloco.

```
<div>
  <p>Parágrafo 1</p>
  <p>Parágrafo 2</p>
</div>
```

Elementos de Linha:

****: O elemento **** é um elemento de linha, o que significa que ele ocupa apenas o espaço necessário dentro do conteúdo pai e não inicia em uma nova linha. Ele é frequentemente usado para aplicar estilos específicos a partes do texto ou agrupar elementos inline para aplicar um estilo ou comportamento específico.

```
<p>Este é um texto normal com um <span style="color: red;">trecho destacado</span>.</p>
```

Em resumo, **<div>** é um elemento de nível de bloco usado para agrupar e estruturar conteúdo, enquanto **** é um elemento de linha usado para aplicar estilos ou agrupar elementos em linha. Ambos são fundamentais para a organização e apresentação do conteúdo em páginas da web.

SELETORES

Os seletores CSS são utilizados para selecionar os elementos HTML aos quais desejamos aplicar estilos. Eles definem quais elementos um conjunto de regras CSS se aplica. Existem diferentes tipos de seletores, cada um com uma sintaxe específica e um nível de especificidade que determina a ordem em que as regras são aplicadas.

Aqui estão alguns exemplos de seletores CSS:

- **Seletor de tipo:** Seleciona os elementos com base em seu tipo, ou seja, os elementos das tags.
- **Seletor de classe:** Seleciona os elementos com base em uma classe atribuída a eles. O seletor de classe é representado pelo caractere ., seguido pelo nome da classe. Um elemento pode pertencer a várias classes / grupos de estilização.

Para aplicar duas classes CSS a um único elemento, podemos utilizar a técnica de aninhar classes. Isso significa que um elemento pode ter mais de uma classe atribuída a ele, separadas por espaço. Quando isso acontece, o elemento receberá os estilos das duas classes.

```
<div class="classe1 classe2">Conteúdo do elemento</div>
```

SELETORES

No contexto do CSS, a regra **.class1.class2 {color: white;}** é um exemplo de seleção de elementos que possuem duas classes simultaneamente.

Quando duas classes são combinadas sem espaço entre elas, isso indica que a regra se aplica apenas aos elementos que possuem ambas as classes atribuídas a eles. Nesse caso, a cor do texto desses elementos será definida como branco.

```
<p class="classe1 classe2">Este parágrafo possui as classes classe1 e classe2.</p>
<p class="classe1">Este parágrafo possui apenas a classe classe1.</p>
<p class="classe2">Este parágrafo possui apenas a classe classe2.</p>
```

A regra **.class1.class2 { color: white; }** se aplicará apenas ao primeiro parágrafo, pois é o único elemento que possui ambas as classes classe1 e classe2. A cor do texto desse parágrafo será definida como branco.

Os outros dois parágrafos não possuem ambas as classes, portanto, a regra não se aplica a eles. A cor do texto desses parágrafos será determinada por outras regras de estilo ou pelo estilo padrão do navegador.

- **Seletor de ID:** Seleciona um elemento com base em um ID único atribuído a ele. O seletor de ID é representado pelo caractere #, seguido pelo nome do ID.

- **Seletor descendente:** Seleciona elementos que são descendentes diretos ou indiretos de um elemento específico. O seletor descendente é representado pelo espaço entre dois seletores. Por exemplo, o seletor `ul li` seleciona todos os elementos `` que são descendentes diretos ou indiretos de um elemento ``.

A regra `#div-conteudo-4-7 div {}` em CSS é um seletor que seleciona todos os elementos `<div>` que são filhos (descendentes diretos ou indiretos) de um elemento com o ID `div-conteudo-4-7`.

O seletor `#div-conteudo-4-7` seleciona um elemento com o ID específico `div-conteudo-4-7`. Em seguida, o seletor `div` seleciona todos os elementos `<div>` que são descendentes diretos ou indiretos desse elemento com o ID `div-conteudo-4-7`.

A regra `#div-conteudo-4-7 div {}` será aplicada a todos os três elementos `<div>` dentro do elemento com o ID `div-conteudo-4-7`. Qualquer estilo definido dentro dessa regra será aplicado a esses elementos.

É importante notar que a regra `#div-conteudo-4-7 div {}` é um seletor específico que se aplica apenas a elementos dentro do elemento com o ID `div-conteudo-4-7`.

- **Seletor de filho direto:** Seleciona elementos que são filhos diretos de um elemento específico. O seletor de filho direto é representado pelo caractere `>` entre dois seletores. Por exemplo, o seletor `ul > li` seleciona todos os elementos `` que são filhos diretos de um elemento ``.

A regra `#div-conteudo-4-7 > div {}` em CSS é um seletor que seleciona todos os elementos `<div>` que são filhos diretos de um elemento com o ID `div-conteudo-4-7`. O sinal `>` indica que a seleção deve ser aplicada apenas aos elementos que são filhos diretos do elemento pai.

A regra `#div-conteudo-4-7 > div {}` será aplicada apenas ao primeiro e ao terceiro elemento `<div>` dentro do elemento com o ID `div-conteudo-4-7`. Isso ocorre porque esses dois elementos são filhos diretos do elemento pai. O segundo elemento `<div>` é um descendente indireto, pois está dentro de outro elemento `<div>`. Portanto, a regra não se aplica a ele.

Qualquer estilo definido dentro dessa regra será aplicado apenas aos elementos `<div>` que são filhos diretos do elemento com o ID `div-conteudo-4-7`.

- **Seletor de irmão adjacente:** Seleciona o próximo irmão adjacente de um elemento específico. O seletor de irmão adjacente é representado pelo caractere + entre dois seletores. Por exemplo, o seletor h1 + p seleciona o próximo elemento <p> que é um irmão adjacente de um elemento <h1>.

Esses são apenas alguns exemplos de seletores CSS. Existem muitos outros tipos de seletores, como seletores de atributo, **seletores de pseudo-classe** e seletores de pseudo-elemento, que permitem selecionar elementos com base em características adicionais ou estados específicos.

As **pseudo-classes** em CSS são palavras-chave adicionadas aos seletores para especificar um estado especial do elemento selecionado. Elas permitem aplicar estilos a elementos com base em fatores externos, como o histórico de navegação, o status do conteúdo ou a interação do usuário.

A **pseudo-classe :hover** é usada para aplicar um estilo a um elemento quando o usuário passa o cursor sobre ele. Além da pseudo-classe :hover, existem várias outras pseudo-classes em CSS, como :active para elementos que estão sendo ativados, :visited para links visitados, :focus para elementos que estão com foco, entre outras.

É importante mencionar que as pseudo-classes são diferentes dos pseudo-elementos. Os **pseudo-elementos** são usados para estilizar partes específicas de um elemento, enquanto as pseudo-classes se aplicam ao elemento como um todo.

É importante entender a especificidade dos seletores CSS, pois ela determina a ordem em que as regras CSS são aplicadas quando há conflitos. Quanto mais específico for o seletor, maior será sua prioridade. Por exemplo, um seletor de ID é mais específico do que um seletor de classe.

Continuando com a página de internet que criamos, para iniciar a compreensão de HTML e CSS, conseguimos identificar um padrão de como os nossos elementos HTML são ilustrados no nosso navegador.

Repare na nossa página que os nossos **elementos possuem um formato de retângulos ou caixas**, e podemos colocar retângulos dentro de retângulos, e a forma que eles terão, o espaçamento entre eles, margens, posicionamento se torna mais fácil quando começamos a entender esse conceito.

Essencialmente, o que iremos manipular são retângulos.

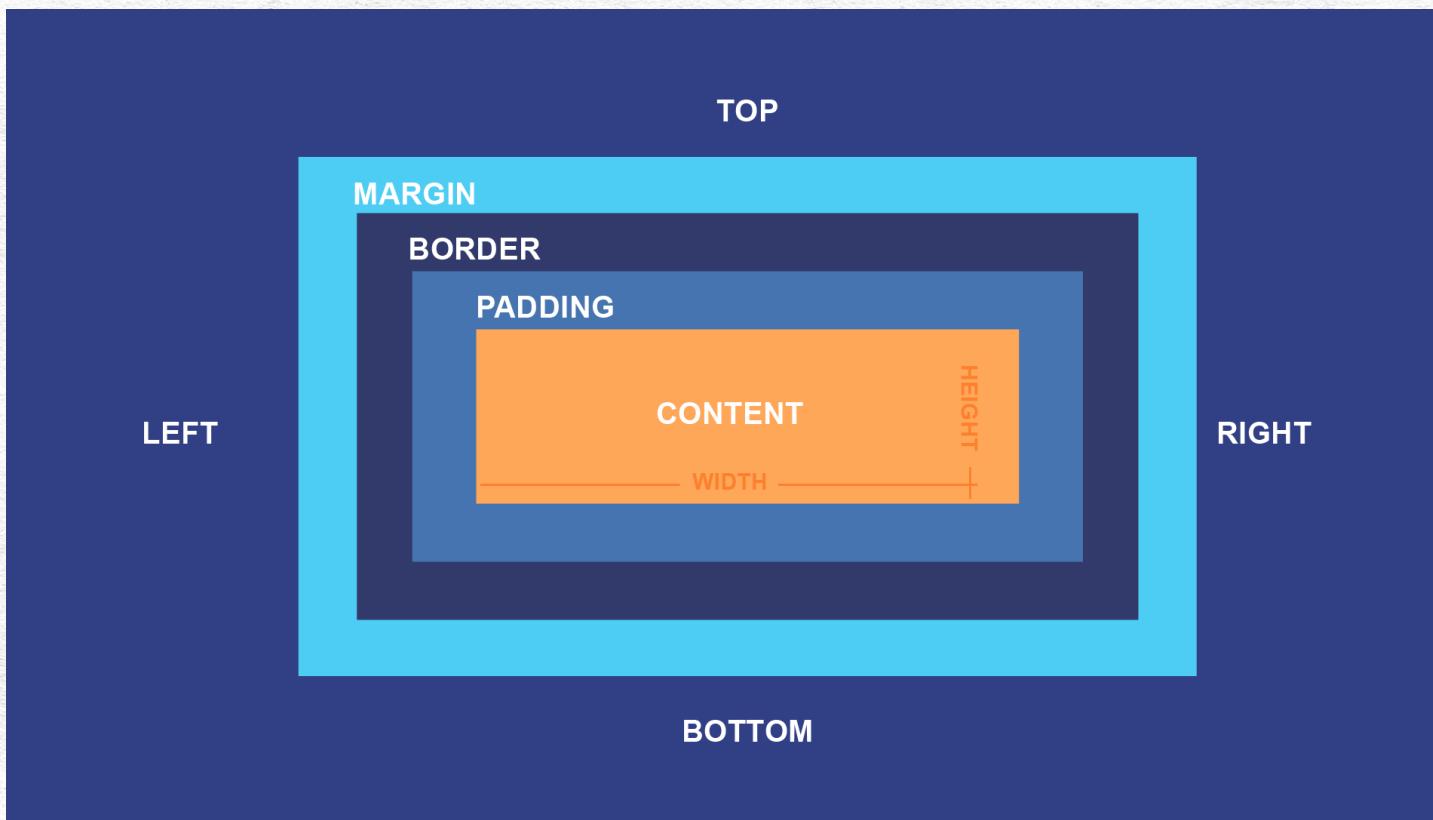


BOX MODEL

Box Model é um conceito fundamental no desenvolvimento web que descreve a estrutura de uma caixa que envolve cada elemento HTML. Essa caixa é composta por quatro partes principais: conteúdo, preenchimento (padding), borda (border) e margem (margin). O modelo de caixa é utilizado para determinar o tamanho, posicionamento e espaçamento dos elementos em uma página.

- **Conteúdo (Content):** É a parte interna da caixa e representa o espaço ocupado pelo texto, imagens ou outros elementos HTML.
- **Preenchimento (Padding):** É uma área transparente que fica entre o conteúdo e a borda da caixa. O preenchimento é usado para criar espaço entre o conteúdo e a borda, proporcionando um espaçamento interno.
- **Borda (Border):** É uma linha ou traço que envolve o conteúdo e o preenchimento. A borda pode ser estilizada com diferentes estilos, cores e espessuras.
- **Margem (Margin):** É uma área transparente que fica fora da borda da caixa. A margem é usada para criar espaço entre a caixa do elemento e outros elementos adjacentes, proporcionando um espaçamento externo.

Essas partes do modelo de caixa afetam o dimensionamento e o posicionamento dos elementos na página. Por exemplo, ao definir a largura e altura de um elemento, você está especificando o tamanho do conteúdo, excluindo o preenchimento, a borda e a margem. As **propriedades CSS como width, height, padding, border e margin** são usadas para controlar essas partes do modelo de caixa.



AS PROPRIEDADES CSS : PADDING, BORDER E MARGIN.

- **Padding:** O padding é uma propriedade que define o espaçamento entre o conteúdo de um elemento e sua borda. Ele cria uma área transparente ao redor do conteúdo. O valor do padding pode ser definido individualmente para cada lado (superior, direito, inferior e esquerdo) ou em uma única propriedade usando a notação abreviada.

Por exemplo, padding: 10px; define um espaçamento de 10 pixels em todos os lados do elemento.

- **Border:** A propriedade border define a borda de um elemento. Ela pode ser utilizada para estilizar a borda do elemento com uma linha ou traço. A propriedade border possui várias subpropriedades, como **border-width** (espessura da borda), **border-style** (estilo da borda) e **border-color** (cor da borda).

Por exemplo, border: 1px solid black; define uma borda sólida de 1 pixel de espessura e cor preta.

- **Margin:** A propriedade margin define o espaçamento externo de um elemento. Ela cria uma área transparente ao redor da borda do elemento, criando espaço entre o elemento e outros elementos vizinhos. O valor do margin pode ser definido individualmente para cada lado ou em uma única propriedade usando a notação abreviada.

Por exemplo, margin: 10px; define um espaçamento de 10 pixels em todos os lados do elemento.

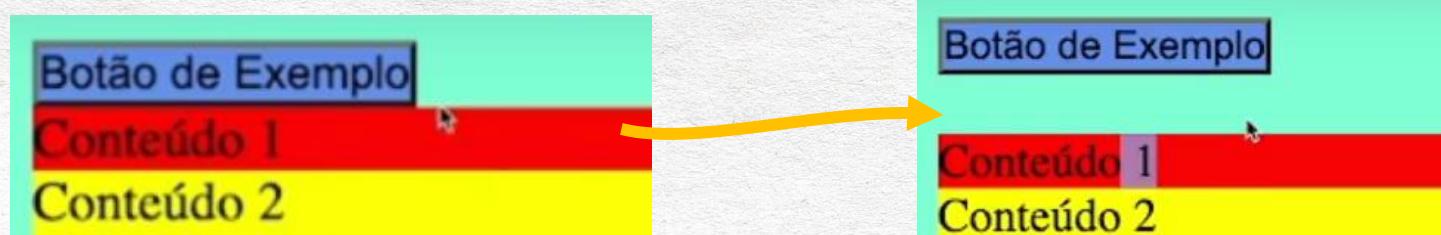
Por exemplo, se alterarmos a propriedade **padding** no arquivo CSS do nosso botão, e atribuirmos o valor de 0, podemos observar que retiramos o preenchimento entre conteúdo e borda.

```
button {  
    background-color: cornflowerblue;  
    padding: 0;  
}
```



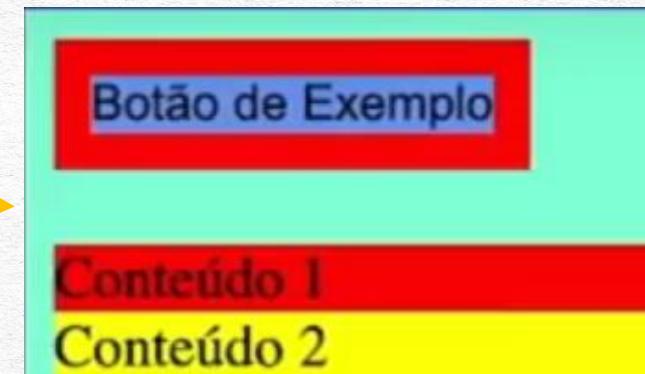
Se adicionarmos na propriedade **margin-bottom** do nosso botão um valor de 20pixels, verificamos que houve uma alteração entre o botão e o elemento abaixo dele.

```
button {  
    background-color: cornflowerblue;  
    padding: 0;  
    margin-bottom: 20px;  
}
```



Modificando a propriedade **border** do botão, alterando suas propriedades:

```
button {  
    background-color: cornflowerblue;  
    padding: 0;  
    margin-bottom: 20px;  
    border: 10px solid red;  
}
```



Essas propriedades são essenciais para criar layouts responsivos e controlar o posicionamento dos elementos na página. Elas permitem ajustar o tamanho dos elementos, adicionar espaçamento interno e externo, e estilizar as bordas dos elementos de acordo com o design desejado.

Relembrando alguns conceitos vistos anteriormente, as **propriedades de CSS** são recursos que nos permitem estilizar e controlar a aparência dos elementos em uma página web. Elas nos ajudam a definir características como tamanho, cor, espaçamento, posicionamento e animações dos elementos.

Para entender como os elementos são renderizados na página, é importante conhecer o conceito de box model. O **box model** é uma representação dos elementos como caixas retangulares. Cada caixa é composta por quatro partes: **conteúdo, preenchimento, borda e margem**.

- A área de conteúdo é onde o texto e outros elementos são exibidos. É o espaço ocupado pelo conteúdo real do elemento, como texto, imagens ou vídeos.
- O preenchimento é uma área transparente ao redor do conteúdo, criando um espaçamento interno entre o conteúdo e a borda do elemento.
- A borda é uma linha que envolve o conteúdo e o preenchimento. Ela pode ter diferentes estilos, espessuras e cores.
- A margem é uma área transparente que fica fora da borda. Ela cria um espaçamento externo entre o elemento e outros elementos vizinhos.

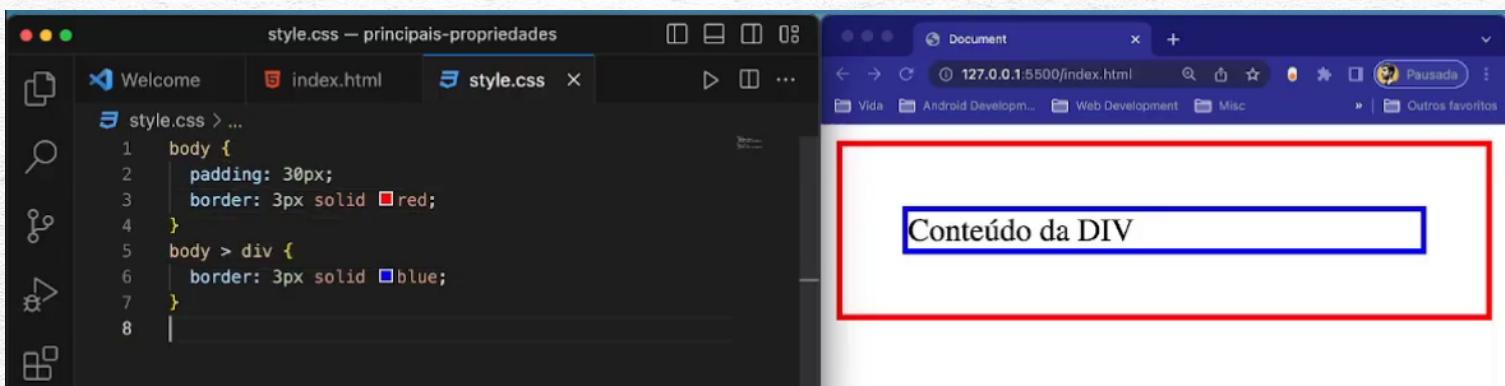
Essas quatro partes juntas formam o box model de um elemento. O tamanho total do elemento é calculado levando em consideração o tamanho do conteúdo, o preenchimento, a borda e a margem.

Ao utilizar propriedades CSS como width, height, padding, border e margin, podemos controlar o tamanho, espaçamento e aparência dos elementos de acordo com nossas necessidades. Essas propriedades são essenciais para criar layouts responsivos e estilizar os elementos de forma personalizada.

Vamos iniciar com a estrutura do nosso arquivo index.html:

```
index.html > html > head > link
1  <!DOCTYPE html>
2  <html lang="en">
3  | <head>
4  | | <meta charset="UTF-8" />
5  | | <meta name="viewport" content="width=device-width,
6  | | <title>Document</title>
7  | | <link rel="stylesheet" href=".style.css" />
8  | </head>
9  | <body>
10 | | <div>Conteúdo da DIV</div>
11 | </body>
12 </html>
13
```

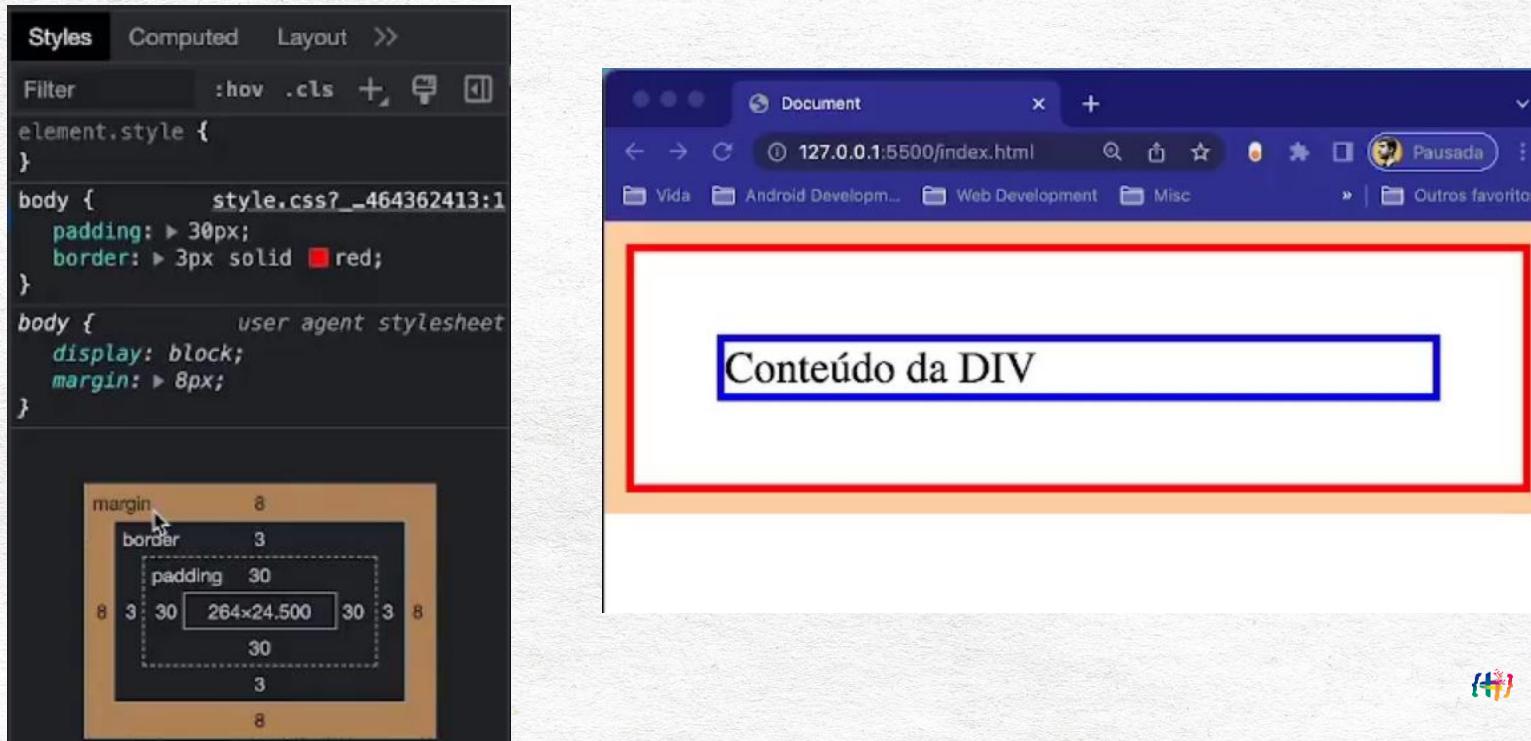
E adicionaremos algumas estilizações na nossa página, para verificarmos o conceito do box model e de como ele age de tags pai/filho:



O elemento <body> é uma das principais partes de um documento HTML. Ele é responsável por conter todo o conteúdo que será exibido para o usuário em um navegador. O <body> é delimitado pelo par de tags <body> e </body>, e dentro dele você irá adicionar todos os elementos que compõem a estrutura e o conteúdo da sua página.

Então caso existe ainda um espaço fora do nosso body, sabemos que é a sua margem.

Podemos verificar isso utilizando o DevTools do seu navegador:



Agora vamos adicionar mais um elemento em nossa estrutura HTML e observar como está sua estilização.

The screenshot shows a developer setup with three windows:

- index.html — principais-propriedades**: Contains the following HTML code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<meta name="viewport" content="width=device-width,
<title>Document</title>
<link rel="stylesheet" href="./style.css" />
</head>
<body>
<div>Conteúdo da DIV</div>
<div>Div irmã</div>
</body>
</html>
```
- style.css — principais-propriedades**: Contains the following CSS code:

```
body {
  padding: 30px;
  border: 3px solid red;
}
body > div {
  border: 3px solid blue;
  padding: 10px;
  margin: 15px;
  background-color: aqua;
  color: purple;
}
```
- Document**: A browser window showing the rendered HTML. It features two blue-bordered boxes with purple text:
 - The top box contains the text "Conteúdo da DIV".
 - The bottom box contains the text "Div irmã".

Observando o código acima, podemos ver que a estilização é a mesma entre os dois elementos divs, pois não adicionamos nenhum seletor mais específico e nem outra estilização para a div irmã.

- **Propriedade border-radius:** O border-radius é uma propriedade CSS que define o raio dos cantos de um elemento. Com essa propriedade, é possível criar elementos com cantos arredondados, dando um aspecto mais suave e estético ao design da página.

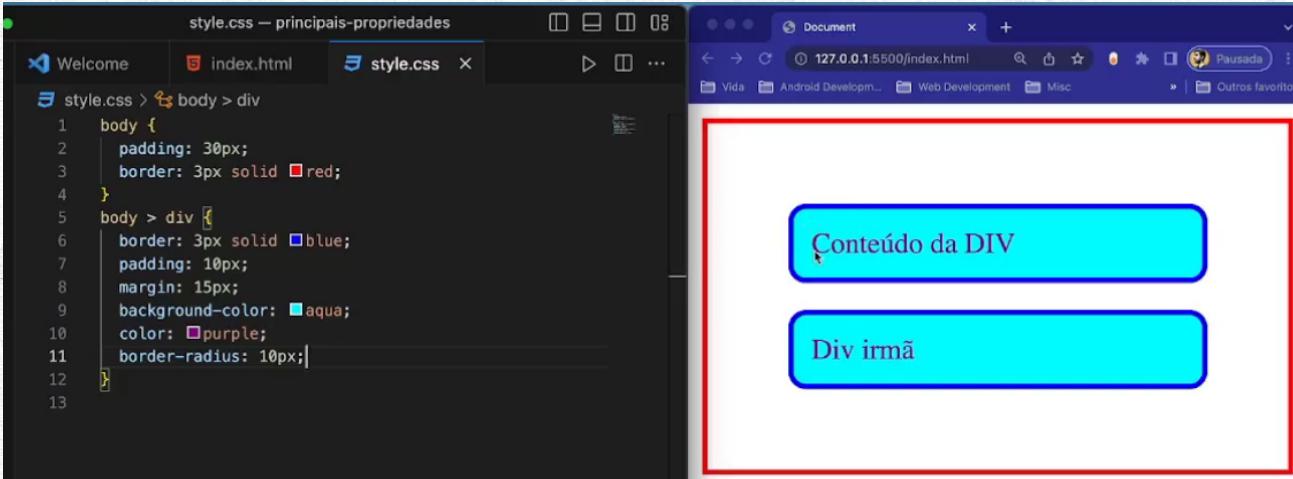
O valor pode ser especificado de diferentes formas, permitindo criar diferentes tipos de formas nos cantos do elemento. Alguns exemplos de valores que podem ser utilizados são:

- **Um único valor:** define o raio dos quatro cantos do elemento com o mesmo valor.
- **Dois valores:** define o raio dos cantos opostos do elemento separadamente. O primeiro valor define o raio dos cantos superiores esquerdo e direito, e o segundo valor define o raio dos cantos inferiores esquerdo e direito.
- **Três valores:** define o raio dos cantos de forma individual. O primeiro valor define o raio do canto superior esquerdo, o segundo valor define o raio dos cantos superior direito e inferior esquerdo, e o terceiro valor define o raio do canto inferior direito.
- **Quatro valores:** define o raio de cada canto individualmente. O primeiro valor define o raio do canto superior esquerdo, o segundo valor define o raio do canto superior direito, o terceiro valor define o raio do canto inferior direito, e o quarto valor define o raio do canto inferior esquerdo.

Além disso, a propriedade border-radius também pode receber valores em porcentagem, permitindo criar elementos com cantos arredondados de forma proporcional ao tamanho do elemento.

É importante ressaltar que a propriedade border-radius pode ser combinada com outras propriedades relacionadas aos cantos, como border-top-left-radius, border-top-right-radius, border-bottom-right-radius e border-bottom-left-radius. Essas propriedades permitem definir o raio de cada canto individualmente, oferecendo ainda mais flexibilidade na criação de formas personalizadas nos cantos dos elementos.

Exemplos da utilização da propriedade border-radius:

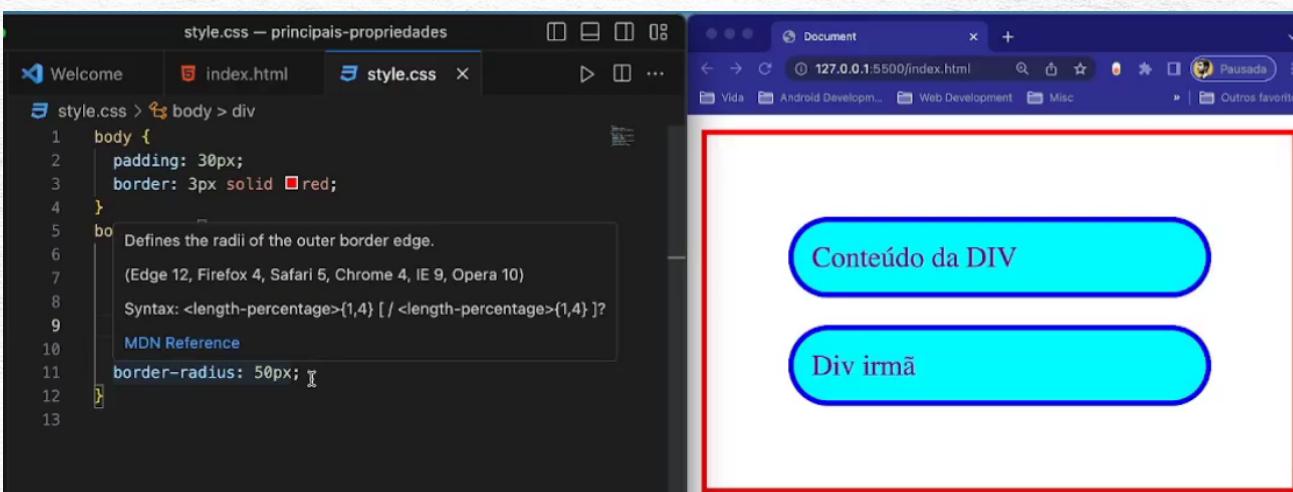


The screenshot shows a code editor and a browser window. The code editor displays the following CSS:

```
style.css — principais-propriedades
Welcome index.html style.css
style.css > body > div
1 body {
2   padding: 30px;
3   border: 3px solid red;
4 }
5 body > div {
6   border: 3px solid blue;
7   padding: 10px;
8   margin: 15px;
9   background-color: aqua;
10  color: purple;
11  border-radius: 10px;
12 }
13
```

The browser window shows a single

element with a red border and rounded corners containing the text "Conteúdo da DIV" and "Div irmã".



The screenshot shows a code editor and a browser window. The code editor displays the same CSS as the previous screenshot, but with a tooltip for the border-radius property:

```
style.css — principais-propriedades
Welcome index.html style.css
style.css > body > div
1 body {
2   padding: 30px;
3   border: 3px solid red;
4 }
5 body > div {
6   border: 3px solid blue;
7   padding: 10px;
8   margin: 15px;
9   background-color: aqua;
10  color: purple;
11  border-radius: 50px; // Defines the radii of the outer border edge.
12  // Syntax: <length-percentage>{1,4} [ / <length-percentage>{1,4} ]?
13  MDN Reference
```

The browser window shows two

elements with red borders and rounded corners, containing the text "Conteúdo da DIV" and "Div irmã".



Introdução do Projeto – PizzaHash

Neste projeto, vamos desenvolver uma **Landing Page** para a pizzaria **PizzaHash**, utilizando **HTML** e **CSS**. Este projeto irá aplicar diversos conceitos de **desenvolvimento web** que vimos ao longo do módulo, incluindo a criação de uma estrutura básica de página, organização do conteúdo e aplicação de estilos responsivos.

A página será composta por várias seções interativas, como:

- **Header (Cabeçalho):** Incluindo o logotipo, o menu de navegação e ícones para funcionalidades adicionais (como o carrinho de compras e login).
- **Home:** A seção inicial com uma saudação e uma chamada para ação que incentiva o usuário a fazer um pedido.
- **Sobre (About):** Explicação sobre a pizzaria, seus diferenciais e o que a torna única.
- **Menu:** Apresentação das pizzas disponíveis no cardápio, com suas descrições e preços.
- **Promoções (Products):** Exibição de combos e promoções, com opções atrativas para os clientes.
- **Avaliações (Review):** Depoimentos de clientes satisfeitos com a pizzaria.
- **Footer (Rodapé):** Informações adicionais sobre a empresa, links úteis e redes sociais.

Através deste projeto, iremos aplicar práticas como:

- Uso de **semântica HTML** para uma estrutura de conteúdo bem organizada.
- **CSS para layout e estilo**, incluindo uso de Flexbox ou Grid para organizar o conteúdo de forma responsiva.
- Personalização e interação com ícones, utilizando a biblioteca **Ionicons** para uma interface mais moderna e funcional.

Esse projeto não só permitirá a criação de uma página visualmente atraente, mas também é uma ótima oportunidade para aplicar os conceitos de estrutura, responsividade e design que foram explorados no módulo de **desenvolvimento web**.

Nesta primeira aula, vamos aprender como criar a estrutura básica de um arquivo HTML no **VS Code**, configurar um arquivo de estilo CSS externo e importar uma fonte do **Google Fonts** para personalizar o texto da nossa página.

Passo 1: Criando a Estrutura Básica de HTML

A estrutura básica de um documento HTML é composta por algumas tags essenciais que ajudam a definir o conteúdo e sua organização. Vamos criar um arquivo HTML básico no **VS Code**:

- Abra o **VS Code** e crie uma nova pasta para o seu projeto.
- Dentro dessa pasta, crie um arquivo chamado **index.html**. E utilize o atalho ! + Enter para criar a estrutura básica de um documento HTML e vamos aplicar a importação do CSS e modificar o title da nossa página.

Aqui está a estrutura básica do nosso documento HTML:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Landing Page PizzaHash</title>
    <link rel="stylesheet" href="style.css">
  </head>
  <body> ...
  </body>
</html>
```



Passo 2: Criando o Arquivo de Estilos (CSS)

Agora que criamos o arquivo HTML, vamos adicionar os estilos utilizando o **CSS**. Siga os passos abaixo:

- Na mesma pasta do arquivo HTML, crie um arquivo chamado **style.css**.
- No arquivo CSS, você aplicará o estilo à sua página HTML.

Passo 3: Importar a Fonte do Google Fonts no CSS

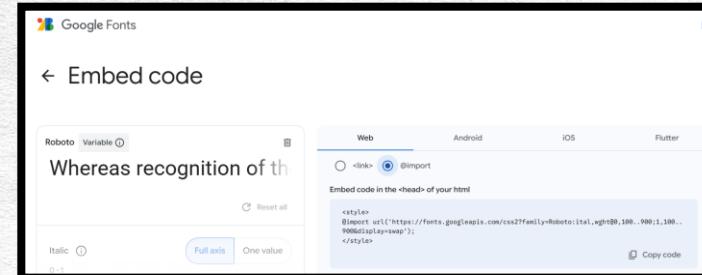
No arquivo style.css, vamos importar uma fonte do Google Fonts, no caso, a fonte **Roboto**. A importação será feita no início do arquivo CSS, para garantir que todos os estilos sejam aplicados corretamente.

Adicione o seguinte código no arquivo style.css:

```
/* Importação da fonte Roboto */  
@import url('https://fonts.googleapis.com/css2?family=Roboto:wght@400;700&display=swap');
```

Explicação do Código:

- **@import URL**: A linha @import é utilizada para importar a fonte **Roboto** do Google Fonts. Você pode modificar a URL para importar outras fontes ou estilos (por exemplo, **bold**, **italics**, etc.). A propriedade display=swap ajuda a garantir que o texto será renderizado o mais rápido possível, mesmo se a fonte ainda não tiver sido carregada.

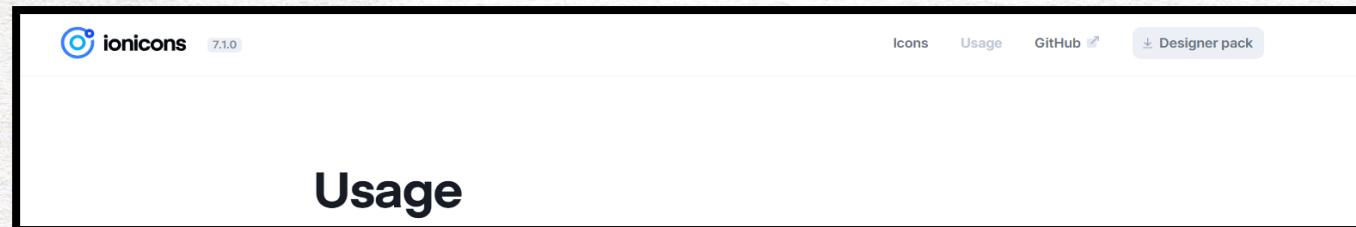


Agora vamos aprender a importar a biblioteca **Ionicons** e a usar seus ícones em um projeto HTML. O Ionicons oferece uma grande variedade de ícones para ser usados em sites, proporcionando uma maneira fácil e rápida de melhorar o design da sua página.

Passo 1: Importando o Ionicons no seu Projeto

- Para começar, vamos importar o link da biblioteca **Ionicons** no arquivo HTML.

```
<script type="module" src="https://unpkg.com/ionicons@7.1.0/dist/ionicons/ionicons.esm.js"></script>
<script nomodule src="https://unpkg.com/ionicons@7.1.0/dist/ionicons/ionicons.js"></script>
</body>
</html>
```



Installation

If you're using Ionic Framework, Ionicons is packaged by default, so no installation is necessary. Want to use Ionicons without Ionic Framework? Place the following `<script>` near the end of your page, right before the closing `</body>` tag, to enable them.

```
<script type="module" src="https://unpkg.com/ionicons@7.1.0/dist/ionicons/ionicons.esm.js"></script>
<script nomodule src="https://unpkg.com/ionicons@7.1.0/dist/ionicons/ionicons.js"></script>
```

Basic usage

To use a built-in icon from the Ionicons package, populate the `name` attribute on the `ion-icon` component:

```
<ion-icon name="heart"></ion-icon>
```

Explicação do Código:

- O **script type="module"** carrega a versão mais recente dos ícones (modular), e o **script nomodule** serve para navegadores mais antigos que não suportam módulos.
- Esses links importam a biblioteca de ícones para que possamos utilizá-los no corpo do HTML.

Passo 2: Usando os Ícones no HTML

Agora que você importou a biblioteca **Ionicons**, pode começar a usar os ícones no seu projeto. Para adicionar um ícone ao seu HTML, basta usar a tag <ion-icon> com o nome do ícone que você deseja usar.

Aqui está um exemplo com alguns ícones populares:

```
<ion-icon name="person"></ion-icon>
<ion-icon name="search"></ion-icon>
<ion-icon name="cart"></ion-icon>
<ion-icon name="menu"></ion-icon>
```

Explicação do Código:

- A tag <ion-icon> é usada para representar o ícone.
- A propriedade name dentro da tag define qual ícone será exibido. Por exemplo, name="person" vai exibir o ícone de uma pessoa, e name="search" vai exibir o ícone de uma lupa.

Vamos localizar todos os ícones que vamos utilizar dentro do nosso projeto e deixar comentado dentro do nosso body, para quando formos utilizar já temos a referência do elemento salva.

```
<body>

    <!-- <ion-icon name="person"></ion-icon> Login -->
    <!-- <ion-icon name="cart"></ion-icon> Cart -->
    <!-- <ion-icon name="search"></ion-icon> Lupa -->
    <!-- <ion-icon name="menu"></ion-icon> Menu -->
    <!-- <ion-icon name="heart"></ion-icon> coração -->
    <!-- <ion-icon name="eye"></ion-icon> Olho -->
    <!-- <ion-icon name="star"></ion-icon> Estrela Completa -->
    <!-- <ion-icon name="star-half"></ion-icon> Estrela Metade -->
    <!-- <ion-icon name="logo-facebook"></ion-icon> Facebook -->
    <!-- <ion-icon name="logo-instagram"></ion-icon> instagram -->
    <!-- <ion-icon name="logo-twitter"></ion-icon> Twitter -->
    <!-- <ion-icon name="logo-linkedin"></ion-icon> Link -->

</body>
```

O objetivo dessa aula é criar a estrutura básica do **header** (cabeçalho) e da **home** (página inicial) do site, incluindo links de navegação e elementos de ícones. Vamos organizar a estrutura do HTML, garantindo uma página com um layout funcional e agradável.

Passo 1: Estrutura do Header

O **header** é a área de cabeçalho do site, onde normalmente colocamos o logo, a navegação e ícones de interatividade (como perfil, busca e carrinho).

1. Logo:

- A tag `<a>` é usada para criar um link, no caso, ao clicar no logo, o usuário será redirecionado para o topo da página (isso é definido pelo `href="#"`).
- O logo é inserido com a tag ``, e o alt é utilizado para fornecer uma descrição do conteúdo da imagem caso ela não seja carregada.

2. Navegação:

- A tag `<nav>` é usada para agrupar os links de navegação do site.
- Cada `<a>` é um link de navegação, com um atributo href apontando para um id de seção na página (ex: `#home`, `#about`). Isso permite que, ao clicar em um link, o usuário seja levado diretamente para a seção correspondente.

```
<!-- HEADER -->
<header class="header">
  <!-- Logo -->
  <a href="#" class="logo">
    
  </a>

  <!-- Navegação -->
  <nav class="navbar">
    <a href="#home">home</a>
    <a href="#about">sobre</a>
    <a href="#menu">menu</a>
    <a href="#products">promoções</a>
    <a href="#review">avaliações</a>
  </nav>

  <!-- Ícones -->
  <div class="icons">
    <div><ion-icon name="person"></ion-icon></div>
    <div><ion-icon name="search"></ion-icon></div>
    <div><ion-icon name="cart"></ion-icon></div>
    <div id="menu-btn"><ion-icon name="menu"></ion-icon></div>
  </div>
</header>
```

3. Ícones:

- O bloco de ícones é colocado dentro de um `<div>` com a classe `icons`. Cada ícone é inserido com a tag `<ion-icon>`, especificando o nome do ícone com o atributo `name`. No exemplo, estamos usando ícones para "perfil", "pesquisa", "carrinho" e "menu".
- O ícone do menu (`<ion-icon name="menu"></ion-icon>`) pode ser utilizado para um botão de menu que, normalmente, abre um menu de navegação em dispositivos móveis ou telas menores.

Passo 2: Estrutura da Home

Agora, vamos estruturar a seção **home** (página inicial). O objetivo dessa seção é dar boas-vindas ao usuário e fornecer um botão de ação, como "Faça seu pedido".

Aqui está a estrutura da **home**:

```
<!-- HOME -->
<section class="home" id="home">
  <div class="content">
    <h3>Bem-vindo à PizzaHash!</h3>
    <p>Descubra o sabor autêntico das nossas pizzas artesanais, preparadas com ingredientes frescos e de alta qualidade.
       Venha experimentar uma experiência gastronômica inesquecível!</p>
    <a href="#" class="btn">Faça seu pedido!</a>
  </div>
</section>
```

1. Seção Home:

- A tag <section> é usada para definir uma seção da página. Ela agrupa conteúdos relacionados, como o título, parágrafo e botão de ação.
- O atributo id="home" permite que você faça referência a esta seção de forma fácil no menu de navegação. Por exemplo, o link "home" no header vai direcionar para essa seção da página.

2. Conteúdo da Home:

O conteúdo principal da home está dentro de uma <div> com a classe content. Ela contém:

- Um título <h3> que dá as boas-vindas ao visitante.
- Um parágrafo <p> que descreve a pizzaria e convida o visitante a conhecer mais sobre o que ela oferece.
- Um botão de ação <a> com a classe btn, que direciona o usuário para uma ação, como fazer o pedido.

Resumo da Aula:

- **Header:** Aprendemos a estruturar o cabeçalho de um site com logo, navegação e ícones. Cada link de navegação tem um href que aponta para um ID da seção correspondente.
- **Home:** Criamos a seção inicial da página com um título de boas-vindas, uma breve descrição e um botão de ação para engajar o visitante.

Nesta aula, vamos estruturar a seção **Sobre (About)** de um site, onde explicamos mais sobre a pizzaria, os diferenciais e o que a torna única. Vamos usar uma combinação de texto e imagens para criar uma seção atraente e informativa.

Estrutura do Código HTML para "Sobre (About)"

```
<!-- SOBRE / ABOUT -->
<section class="about" id="about">
  <h1 class="heading"><span>Sobre</span> nós</h1>

  <div class="row">
    <!-- Imagem -->
    <div class="image">
      
    </div>

    <!-- Conteúdo -->
    <div class="content">
      <h3>O que torna a PizzaHash única?</h3>
      <p>Na PizzaHash, não é apenas sobre pizza – é sobre criar experiências memoráveis. Fundada com a paixão por comida de qualidade e um compromisso com a excelência, nossa pizzaria combina tradição e inovação para oferecer o melhor em cada fatia.</p>
      <p>Nossos ingredientes são cuidadosamente selecionados para garantir frescor e sabor. Usamos técnicas de preparo artesanais para criar massas incrivelmente leves e crocantes, e nossos molhos são feitos a partir de tomates frescos e especiarias únicas.</p>
      <a href="#" class="btn">Saiba mais</a>
    </div>
  </div>
</section>
```

1. Seção Sobre (About):

- A tag <section> é usada para agrupar uma parte do conteúdo da página, que neste caso é a seção "Sobre nós".
- O atributo id="about" é importante porque permite que a seção seja facilmente acessada pelos links de navegação (por exemplo, quando o usuário clicar em "sobre" no menu de navegação).

2. Título da Seção:

- O título da seção é envolvido por um <h1>, que indica a importância do título na hierarquia da página.
- O uso de dentro do título permite que a palavra "Sobre" seja estilizada de forma diferente (se necessário).

3. Estrutura em Linha (Row):

- O <div> com a classe row é utilizado para criar uma linha que agrupa a imagem e o conteúdo de texto. Essa estrutura pode ser utilizada em conjunto com o CSS para garantir que a imagem e o texto fiquem alinhados lado a lado ou em um layout responsivo.

4. Imagem:

- A tag é usada para inserir a imagem que representa a pizzaria. A imagem é carregada de um diretório images (como images/sobre.jpg).
- O atributo alt é utilizado para descrever a imagem, caso ela não seja carregada corretamente ou para acessibilidade, permitindo que leitores de tela compreendam o conteúdo da imagem.

5. Conteúdo de Texto:

- Dentro de outro <div> com a classe content, temos o título <h3>, que apresenta a pergunta "O que torna a PizzaHash única?".
- O texto informativo sobre a pizzaria é colocado dentro de <p> (parágrafos). Aqui, o conteúdo destaca a qualidade dos ingredientes e os métodos de preparação usados pela pizzaria.
- O link <a> com a classe btn funciona como um botão de ação, permitindo que os usuários cliquem para obter mais informações (o link ainda está vazio, mas pode ser direcionado para uma página interna ou externa).

6. Finalizando a Seção:

- A estrutura termina com o fechamento das tags correspondentes, como </div> para os containers de imagem e conteúdo, e </section> para encerrar a seção.

Resumo da Aula:

- **Seção Sobre (About):** A estrutura da seção "Sobre" envolve a criação de um título (usando <h1>), uma imagem (usando) e um conteúdo textual explicativo sobre o negócio.
- Usamos a tag <a> para criar um botão de ação que leva o usuário a mais informações, o que torna a seção interativa.
- A estrutura de layout pode ser ajustada no CSS para garantir que a imagem e o conteúdo textual fiquem bem organizados, seja lado a lado ou em um formato mais responsivo.

Vamos estruturar a seção de **Menu** do site, exibindo as pizzas disponíveis com suas informações, como nome, imagem, preço e a opção de adicionar ao carrinho. Utilizaremos a tag <section> para a estrutura e outras tags de HTML para cada item do menu.

1. Estrutura do Código HTML para "Menu"- Início da Seção "Menu":

```
<section class="menu" id="menu">
```

- A tag <section> define que estamos criando uma seção do conteúdo, neste caso, a seção do **menu**.
- O atributo id="menu" serve para que possamos fazer referência a essa seção com links de navegação, como o item "menu" no cabeçalho.

2. Título da Seção:

```
<h1 class="heading">Nosso <span>menu</span> </h1>
```

- O título da seção é envolvido por uma tag <h1>, que é um título principal.
- O uso do permite destacar a palavra "menu" com um estilo diferente, caso necessário.

3. Contêiner de Itens do Menu:

```
<div class="box-container">
```

- O <div class="box-container"> é o contêiner que agrupa todos os itens do menu. O uso dessa classe ajuda a aplicar estilos (como grid ou flexbox) para dispor os itens de forma organizada.

4. Item de Menu (Pizza):

Para cada pizza que queremos exibir, criamos um bloco como o seguinte:

- **Imagen da Pizza:** A tag é usada para inserir a imagem do item do menu, com o atributo src especificando o caminho da imagem (por exemplo, images/pizza_item_1.jpg). O atributo alt descreve o que está na imagem, ajudando na acessibilidade.
- **Nome da Pizza:** O nome da pizza é exibido dentro da tag <h3>, que é usada para subtítulos. Este nome é o título do item no menu.
- **Preço da Pizza:** O preço original da pizza é exibido dentro de <div class="price">, e o preço com desconto é mostrado dentro de uma tag para destacá-lo visualmente, por exemplo, mostrando o preço antigo e o novo.
- **Botão de Ação:** A tag <a> cria o link para adicionar o item ao carrinho. O texto "Adicionar ao carrinho" dentro da tag <a> será clicável, e o atributo href="#" pode ser substituído por uma URL ou uma função que manipule o carrinho de compras.

```
<div class="box">
  
  <h3>Explosão de Pepperoni</h3>
  <div class="price"> R$55.99 <span>60.99</span></div>
  <a href="#" class="btn">Adicionar ao carrinho</a>
</div>
```

5. Repetição de Itens do Menu:

- Cada item do menu é encapsulado em uma `<div class="box">`, e você pode duplicar essa estrutura para adicionar mais itens, como pizzas diferentes. Basta copiar e colar o bloco de código para cada item novo.

Resumo da Aula:

- Seção Menu:** A seção de menu foi dividida em blocos, cada um representando uma pizza com imagem, nome, preço e um botão de ação.
- Estrutura de Layout:** O uso de classes como `box-container` e `box` permite organizar os itens do menu de maneira flexível e esteticamente agradável.
- Botão de Ação:** A tag `<a>` foi usada para criar um botão de "Adicionar ao carrinho", incentivando a interação do usuário com a página.

```
<!-- MENU -->
<section class="menu" id="menu">
  <h1 class="heading">Nosso <span>menu</span></h1>

  <div class="box-container">
    <div class="box">
      
      <h3>Explosão de Pepperoni</h3>
      <div class="price"> R$55.99 <span>60.99</span></div>
      <a href="#" class="btn">Adicionar ao carrinho</a>
    </div>

    <div class="box">
      
      <h3>Aventuras Vegetais</h3>
      <div class="price"> R$45.99 <span>60.99</span></div>
      <a href="#" class="btn">Adicionar ao carrinho</a>
    </div>

    <div class="box">
      
      <h3>Mediterrânea Delícia</h3>
      <div class="price"> R$75.99 <span>80.99</span></div>
      <a href="#" class="btn">Adicionar ao carrinho</a>
    </div>
  </div>
</section>
```

1

```
<div class="box">
  
  <h3>Pomodoro do Campo</h3>
  <div class="price"> R$45.99 <span>50.99</span></div>
  <a href="#" class="btn">Adicionar ao carrinho</a>
</div>

<div class="box">
  
  <h3>Lusitana Colorida</h3>
  <div class="price"> R$55.99 <span>60.99</span></div>
  <a href="#" class="btn">Adicionar ao carrinho</a>
</div>

<div class="box">
  
  <h3>Saborosa Lava</h3>
  <div class="price"> R$85.99 <span>100.99</span></div>
  <a href="#" class="btn">Adicionar ao carrinho</a>
</div>
</div>
</section>
```

2



Vamos estruturar a seção **Products / Promoções** do site, onde são apresentados os combos e promoções. Cada promoção inclui um conjunto de itens como imagens, descrições, estrelas para avaliação, preços e botões de ação para adicionar ao carrinho.

1. Início da Seção "Combos e Promoções":

```
<section class="products" id="products">
```

- A tag `<section>` define a seção da página que contém as promoções ou combos.
- A classe `products` pode ser usada para aplicar estilos à seção, e o `id="products"` permite navegar para essa seção diretamente a partir do cabeçalho ou menu.

2. Título da Seção:

```
<h1 class="heading">Combos e <span>promoções</span> </h1>
```

- O título da seção é uma `<h1>`, com a palavra "promoções" destacada usando a tag ``, o que permite aplicar um estilo específico a essa parte do texto.

3. Contêiner de Itens de Promoções:

```
<div class="box-container">
```

- O `<div class="box-container">` agrupa os itens de promoções. Ele pode ser estilizado com CSS para organizar os itens em uma grid ou utilizando flexbox, dependendo do layout desejado.

4. Estrutura de Cada Item de Promoção (Pizza + Bebida): Cada item de promoção é estruturado da seguinte forma:

```
<div class="box">
  <div class="icons">
    <a href="#menu"><ion-icon name="cart"></ion-icon></a>
    <a href="#"><ion-icon name="heart"></ion-icon></a>
    <a href="#"><ion-icon name="eye"></ion-icon></a>
  </div>

  <div class="image">
    
  </div>

  <div class="content">
    <h3>Pizza + Refrigerante 2L</h3>
    <div class="stars">
      <i><ion-icon name="star"></ion-icon></i>
      <i><ion-icon name="star"></ion-icon></i>
      <i><ion-icon name="star"></ion-icon></i>
      <i><ion-icon name="star"></ion-icon></i>
    </div>
    <div class="price">R$65.00 <span>a partir</span></div>
  </div>
</div>
```

- **Ícones:** A div icons contém três links com ícones que representam diferentes ações (adicionar ao carrinho, favoritos e visualizar detalhes). Utilizamos o ion-icon para exibir ícones.
- **Imagen:** A imagem do item é inserida na tag ``. A propriedade alt descreve a imagem para acessibilidade.
- **Conteúdo do Item:**
 - O nome da promoção é exibido dentro da tag `<h3>`.
 - As estrelas são criadas com os ícones `<ion-icon name="star">`, representando a avaliação da promoção.
 - O preço é exibido dentro da tag `<div class="price">`, e o preço com desconto ou condição especial é exibido utilizando a tag ``.

5.Repetição de Itens de Promoção: Cada item é envolvido por uma `<div class="box">`. Para adicionar mais promoções, basta duplicar essa estrutura e ajustar as informações de cada item.

Resumo da Aula:

- **Estrutura de Promoções:** Cada promoção é composta por uma imagem, nome, avaliação com estrelas, e preço.
- **Ícones Interativos:** Usamos ícones interativos para ações como adicionar ao carrinho e salvar nos favoritos.
- **Organização em Blocos:** Cada item da promoção é envolvido por uma `<div class="box">` para permitir um layout organizado e flexível.

```
<!-- PRODUCTS / PROMOÇÕES -->
<section class="products" id="products">
  <h1 class="heading">Combos e <span>promoções</span></h1>
  <div class="box-container">
    <!-- Item 1 -->
    <div class="box">
      <div class="icons">
        <a href="#menu"><ion-icon name="cart"></ion-icon></a>
        <a href="#"><ion-icon name="heart"></ion-icon></a>
        <a href="#"><ion-icon name="eye"></ion-icon></a>
      </div>

      <div class="image">
        
      </div>

      <div class="content">
        <h3>Pizza + Refrigerante 2L</h3>
        <div class="stars">
          <i><ion-icon name="star"></ion-icon></i>
          <i><ion-icon name="star"></ion-icon></i>
          <i><ion-icon name="star"></ion-icon></i>
          <i><ion-icon name="star"></ion-icon></i>
          <i><ion-icon name="star"></ion-icon></i>
        </div>
        <div class="price">R$65.00 <span>a partir</span></div>
      </div>
    </div>
  </div>
</section>
```

1

```
<!-- Item 2 -->
<div class="box">
  <div class="icons">
    <a href="#menu"><ion-icon name="cart"></ion-icon></a>
    <a href="#"><ion-icon name="heart"></ion-icon></a>
    <a href="#"><ion-icon name="eye"></ion-icon></a>
  </div>

  <div class="image">
    
  </div>

  <div class="content">
    <h3>Pizza + Refrigerante 350ml</h3>
    <div class="stars">
      <i><ion-icon name="star"></ion-icon></i>
      <i><ion-icon name="star"></ion-icon></i>
      <i><ion-icon name="star"></ion-icon></i>
      <i><ion-icon name="star"></ion-icon></i>
      <i><ion-icon name="star"></ion-icon></i>
    </div>
    <div class="price">R$50.00 <span>a partir</span></div>
  </div>
</div>
```

2

```
<!-- Item 3 -->
<div class="box">
  <div class="icons">
    <a href="#menu"><ion-icon name="cart"></ion-icon></a>
    <a href="#"><ion-icon name="heart"></ion-icon></a>
    <a href="#"><ion-icon name="eye"></ion-icon></a>
  </div>

  <div class="image">
    
  </div>

  <div class="content">
    <h3>Pizza + Refrigerante 600ml</h3>
    <div class="stars">
      <i><ion-icon name="star"></ion-icon></i>
      <i><ion-icon name="star"></ion-icon></i>
      <i><ion-icon name="star"></ion-icon></i>
      <i><ion-icon name="star"></ion-icon></i>
      <i><ion-icon name="star-half"></ion-icon></i>
    </div>
    <div class="price">R$55.00 <span>a partir</span></div>
  </div>
</div>
</section>
```

3



Vamos criar a seção **Review / Avaliações** do site de pizzaria, onde são exibidos os depoimentos de clientes com suas respectivas avaliações. Essa seção é fundamental para dar credibilidade ao negócio e permitir que novos clientes vejam as opiniões dos outros.

1.Início da Seção "Avaliações":

```
<section class="review" id="review">
```

- A tag <section> define a seção de avaliações do site.
- O id="review" permite a navegação direta até essa seção.
- A classe review pode ser usada para aplicar estilos específicos à seção de avaliações.

2.Título da Seção:

```
<h1 class="heading">O que nossos <span>clientes dizem</span></h1>
```

O título é <h1>, destacando a parte "clientes dizem" com a tag , que pode ser estilizada com CSS.

3.Contêiner de Depoimentos:

```
<div class="box-container">
```

A <div class="box-container"> serve para agrupar as caixas de avaliação. Esta classe pode ser usada para aplicar estilos de layout, como Flexbox ou Grid.



4.Estrutura de Cada Depoimento: Cada avaliação segue a mesma estrutura:

```
<div class="box">
  <p>Depoimento do cliente...</p>
  
  <h3>Nome do cliente</h3>
  <div class="stars">
    <i><ion-icon name="star"></ion-icon></i>
    <i><ion-icon name="star"></ion-icon></i>
    <i><ion-icon name="star"></ion-icon></i>
    <i><ion-icon name="star"></ion-icon></i>
    <i><ion-icon name="star-half"></ion-icon></i>
  </div>
</div>
```

- **Depoimento:** O texto do cliente é exibido dentro da tag `<p>`. Ele descreve a experiência com a pizzaria.
- **Imagen:** A foto do cliente é exibida com a tag ``, onde a classe user pode ser estilizada para exibir a imagem de forma adequada.
- **Nome do Cliente:** O nome do cliente é exibido dentro da tag `<h3>`.
- **Estrelas:** A avaliação é feita com estrelas, utilizando os ícones do ion-icon. Cada `<i><ion-icon name="star"></ion-icon></i>` representa uma estrela cheia, enquanto o `<ion-icon name="star-half"></ion-icon>` é uma estrela meio cheia, indicando uma avaliação parcial.

5. Repetição de Depoimentos: Cada depoimento é envolvido por uma <div class="box">. Para adicionar mais avaliações, basta duplicar esse bloco e ajustar o conteúdo (nome, foto e texto).

Resumo da Aula:

- **Depoimentos de Clientes:** Cada depoimento é apresentado com uma imagem do cliente, seu nome, uma descrição e a avaliação por estrelas.
- **Estrelas de Avaliação:** As estrelas são feitas com ícones que representam a pontuação do cliente. As estrelas inteiras indicam a nota máxima, e as estrelas meio cheias indicam avaliações parciais.
- **Organização em Caixas:** Cada avaliação é organizada em uma "caixa", o que facilita a aplicação de estilos visuais e organização do layout.

```
<!-- REVIEW / AVALIAÇÕES -->
<section class="review" id="review">
  <h1 class="heading">O que nossos <span>clientes dizem</span></h1>
  <div class="box-container">
    <!-- Avaliação 1 -->
    <div class="box">
      <p>Lorem ipsum dolor sit amet consectetur adipisicing elit. Animi nulla sit libero nemo fuga sequi nobis? Necessitatibus aut laborum, nisi quas eaque laudantium consequuntur iste ex aliquam minus vel? Nemo.</p>
      
      <h3>Alon Pinheiro</h3>
      <div class="stars">
        <i><ion-icon name="star"></ion-icon></i>
        <i><ion-icon name="star"></ion-icon></i>
        <i><ion-icon name="star"></ion-icon></i>
        <i><ion-icon name="star"></ion-icon></i>
        <i><ion-icon name="star-half"></ion-icon></i>
      </div>
    </div>
  </div>
```

1

```
<!-- Avaliação 3 -->
<div class="box">
  <p>Lorem ipsum dolor sit amet consectetur adipisicing elit. Animi nulla sit libero nemo fuga sequi nobis? Necessitatibus aut laborum, nisi quas eaque laudantium consequuntur iste ex aliquam minus vel? Nemo.</p>
  
  <h3>Daniel Porto</h3>
  <div class="stars">
    <i><ion-icon name="star"></ion-icon></i>
    <i><ion-icon name="star"></ion-icon></i>
    <i><ion-icon name="star"></ion-icon></i>
    <i><ion-icon name="star-half"></ion-icon></i>
  </div>
</div>
</div>
```

3

```
<!-- Avaliação 2 -->
<div class="box">
  <p>Lorem ipsum dolor sit amet consectetur adipisicing elit. Animi nulla sit libero nemo fuga sequi nobis? Necessitatibus aut laborum, nisi quas eaque laudantium consequuntur iste ex aliquam minus vel? Nemo.</p>
  
  <h3>João Lira</h3>
  <div class="stars">
    <i><ion-icon name="star"></ion-icon></i>
    <i><ion-icon name="star"></ion-icon></i>
    <i><ion-icon name="star"></ion-icon></i>
    <i><ion-icon name="star"></ion-icon></i>
    <i><ion-icon name="star"></ion-icon></i>
  </div>
</div>
```

2



Vamos construir a seção **Footer / Rodapé** do site de pizzaria, que é crucial para fornecer informações adicionais, como links úteis, formas de contato e redes sociais. O rodapé geralmente fica fixo na parte inferior da página e contém informações importantes sobre a empresa.

1. Estrutura do Rodapé: O rodapé é representado pela tag <footer class="footer">. Ele contém três colunas com informações e links úteis.

2. Coluna 1: Informações Institucionais:

- Título:** O nome da pizzaria, com uma estilização em "Pizza" e "Hash".
- Links:** Links importantes, como "Sobre nós", "Menu", "Política de Privacidade", e "Termos de Serviço".

```
<div class="footer-col">
    <h4 class="heading-footer">Pizza<span>Hash</span></h4>
    <ul>
        <li><a href="#about">Sobre nós</a></li>
        <li><a href="#menu">Menu</a></li>
        <li><a href="#">Política de Privacidade</a></li>
        <li><a href="#">Termos de Serviços</a></li>
    </ul>
</div>
```



3.Coluna 2: Ajuda ao Cliente:

- **Título:** "Obtenha Ajuda" para questões como suporte e serviços.
- **Links:** Links para FAQ, devoluções e opções de pagamento, fornecendo ajuda e informações adicionais.

```
<div class="footer-col">
    <h4 class="heading-footer">Obtenha <span>Ajuda</span></h4>
    <ul>
        <li><a href="#">FAQ</a></li>
        <li><a href="#">Devoluções</a></li>
        <li><a href="#">Opções de Pagamentos</a></li>
    </ul>
</div>
```

4. Coluna 3: Redes Sociais:

```
<div class="footer-col">
  <h4 class="heading-footer">Siga <span>-nos</span></h4>
  <div class="social-links">
    <a href="#"><ion-icon name="logo-facebook"></ion-icon></a>
    <a href="#"><ion-icon name="logo-instagram"></ion-icon></a>
    <a href="#"><ion-icon name="logo-linkedin"></ion-icon></a>
    <a href="#"><ion-icon name="logo-twitter"></ion-icon></a>
  </div>
</div>
```

- **Título:** "Siga-nos" com links para as redes sociais.
- **Ícones de Redes Sociais:** Usando o ion-icon para adicionar ícones das principais redes sociais, como Facebook, Instagram, LinkedIn e Twitter.

5. Rodapé Inferior:

```
<div class="footer-bottom">  
  <p>&copy; 2024 PizzaHash. Todos os direitos reservados.</p>  
</div>
```

- Esta área contém a mensagem de direitos autorais. O uso de © exibe o símbolo de copyright.
- A classe footer-bottom é usada para aplicar estilos a essa parte final do rodapé.

Resumo da Aula:

- O **rodapé** contém três áreas principais: informações sobre a empresa, links de ajuda ao cliente, e redes sociais.
- Ele termina com uma mensagem de **direitos autorais** para garantir que o conteúdo do site esteja protegido legalmente.
- Usamos **ícones de redes sociais** para promover a interação do usuário com as plataformas sociais da pizzaria.

Estrutura do Código HTML para "Rodapé":

```
<!-- FOOTER / RODAPÉ -->
<footer class="footer">
  <div class="container">
    <!-- Coluna 1: Informações Institucionais -->
    <div class="footer-col">
      <h4 class="heading-footer">Pizza
```

1

```
<!-- Coluna 3: Redes Sociais -->
<div class="footer-col">
  <h4 class="heading-footer">Siga 
```

2



Nesta aula, vamos aprender a criar um estilo global para um site utilizando CSS. Vamos definir variáveis globais, resetar estilos padrões, configurar fontes e melhorar a experiência do usuário.

1. Definindo Variáveis CSS

Podemos definir cores globais utilizando :root, permitindo reutilização e fácil manutenção do código:

```
:root {  
    --main-color: #da190f;  
    --bg: #13131a;  
    --bg-card: #0c0c14;  
    --border: 0.1rem solid rgba(255, 255, 255, 0.3);  
}
```

- --main-color: Define a cor vermelha principal.
- --bg: Define o fundo escuro do site.
- --bg-card: Define a cor de fundo das **cartas/boxes**.
- --border: Define uma borda suítil translúcida.

 **Vantagem das variáveis:** Elas permitem reutilizar cores e estilos ao longo do CSS, facilitando manutenção e personalização.

Dessa forma, se quisermos alterar a cor principal do site, basta modificar o valor de --main-color.

2. Resetando Estilos Padrões

Para garantir que todos os elementos tenham um comportamento uniforme, aplicamos um reset global:

Explicação:

- **margin: 0; padding: 0;** - Remove espaçamentos padrões dos elementos.
- **box-sizing: border-box;** - Garante que a largura inclua padding e borda.
- **outline: none; border: none;** - Remove contornos e bordas padrões.
- **text-decoration: none;** - Remove sublinhados de links.
- **text-transform: capitalize;** - Faz com que cada palavra comece com letra maiúscula.
- **transition: 0.2s linear;** - Adiciona um efeito de transição suave ao interagir com os elementos.

```
* {  
    font-family: 'Roboto', sans-serif;  
    margin: 0;  
    padding: 0;  
    box-sizing: border-box;  
    outline: none;  
    border: none;  
    text-decoration: none;  
    text-transform: capitalize;  
    transition: 0.2s linear;  
}
```

3. Configurações para HTML

Para tornar a rolagem do site mais suave e responsiva, essas configurações melhoram a acessibilidade e responsividade do site:

```
html {  
    font-size: 62.5%; /* Facilita o uso de REM */  
    overflow-x: hidden; /* Remove barra de rolagem horizontal */  
    scroll-padding-top: 9rem; /* Ajusta a rolagem ao usar ancoragem */  
    scroll-behavior: smooth; /* Adiciona rolagem suave */  
}
```

4. Personalizando a Barra de Rolagem

Para um design mais profissional, estilizamos a barra de rolagem:

Explicação:

- **width: 0.8rem;** - Define a largura da barra de rolagem.
- **background: transparent;** - Remove o fundo padrão da barra.
- **background: #fff;** - Define a cor da "pegada" da rolagem.
- **border-radius: 5rem;** - Arredonda a barra para um visual moderno.

```
html::-webkit-scrollbar {  
    width: 0.8rem;  
}  
  
html::-webkit-scrollbar-track {  
    background: transparent;  
}  
  
html::-webkit-scrollbar-thumb {  
    background: #fff;  
    border-radius: 5rem;  
}
```



5. Definindo o Fundo do Site

Para garantir que todo o fundo siga um padrão escuro:

```
body {  
    background: var(--bg);  
}
```

O uso da variável --bg permite uma fácil alteração da cor de fundo global.

6. Configurando as Seções

Todas as seções terão espaçamento adequado para melhor organização:

```
section {  
    padding: 2rem 7%;  
}
```

Isso adiciona espaçamento interno de **2rem na vertical** e **7% na horizontal**.

7. Estilizando os Títulos

Para melhorar a legibilidade dos títulos principais:

```
.heading {  
    text-align: center;  
    color: #fff;  
    text-transform: uppercase;  
    padding-bottom: 3.5rem;  
    font-size: 4rem;  
}
```

E para destacar palavras-chave dentro do título:

```
.heading span {  
    color: var(--main-color);  
    text-transform: uppercase;  
}
```

Isso permite que partes do título tenham a cor principal do site.



Agora vamos iniciar estilização da seção de Cabeçalho (Header):

Explicação das Propriedades

- **background: var(--bg);** - Define o **fundo do cabeçalho** usando a variável --bg, que corresponde a #13131a (um tom escuro).
- **display: flex;** - Ativa o **Flexbox** para organizar os elementos do cabeçalho.
- **align-items: center;** - Alinha os itens verticalmente ao centro.
- **justify-content: space-between;** - Distribui os elementos horizontalmente, deixando um **espaço igual** entre a logo e o menu de navegação.
- **padding: 1.5rem 7%;** - Define **espaçamento interno** (topo e base: 1.5rem, laterais: 7%).
- **border-bottom: var(--border);** - Adiciona uma **linha na parte inferior** do cabeçalho (0.1rem solid rgba(255, 255, 255, 0.3)).
- **position: fixed;** - **Fixa o cabeçalho** no topo da página.
- **top: 0; left: 0; right: 0;** - Garante que o cabeçalho ocupe **toda a largura da tela**.
- **z-index: 1000;** - Garante que o cabeçalho fique acima de outros elementos na página.

```
.header {  
    background: var(--bg);  
    display: flex;  
    align-items: center;  
    justify-content: space-between;  
    padding: 1.5rem 7%;  
    border-bottom: var(--border);  
    position: fixed;  
    top: 0;  
    left: 0;  
    right: 0;  
    z-index: 1000;  
}
```

Estilização da Logo (.header .logo img)

- **height: 6.5rem;** → Define a altura da imagem da logo.
- **border-radius: 100%;** → Transforma a imagem em **formato circular**.

```
.header .logo img {  
    height: 6.5rem;  
    border-radius: 100%;  
}
```

Estilização do Menu (.header .navbar a)

- **margin: 0 1rem;** → Espaçamento horizontal entre os itens do menu.
- **font-size: 1.6rem;** → Define o tamanho da fonte.
- **color: #fff;** → Cor branca para os links do menu.

```
.header .navbar a {  
    margin: 0 1rem;  
    font-size: 1.6rem;  
    color: #fff;  
}
```

Efeito Hover no Menu (.header .navbar a:hover)

- **color: var(--main-color);** → Muda a cor do link ao passar o mouse (#da190f - vermelho).
- **border-bottom: 0.1rem solid var(--main-color);** → Adiciona um **sublinhado vermelho**.
- **padding-bottom: 0.5rem;** → Aumenta o espaçamento entre o texto e o sublinhado.

```
.header .navbar a:hover {  
    color: var(--main-color);  
    border-bottom: 0.1rem solid var(--main-color);  
    padding-bottom: 0.5rem;  
}
```

Estilização do Contêiner de Ícones

- **display: flex;** → Usa **Flexbox** para organizar os ícones horizontalmente.
- **align-items: center;** → Mantém os ícones **centralizados verticalmente**.

```
.header .icons {  
    display: flex;  
    align-items: center;  
}
```

Estilização Individual dos Ícones

- **margin-left: 2rem;** → Cria **espaçamento lateral** entre os ícones para evitar que fiquem colados.

```
.header .icons div {  
    margin-left: 2rem;  
}
```

Estilização dos Ícones do ion-icon

- **color: #fff;** → Define a cor dos ícones como **branca**.
- **cursor: pointer;** → Muda o cursor para **mãozinha** ao passar o mouse.
- **font-size: 2.5rem;** → Aumenta o tamanho dos ícones para **2.5rem**.

```
.header .icons div ion-icon {  
    color: #fff;  
    cursor: pointer;  
    font-size: 2.5rem;  
}
```

Efeito Hover nos Ícones

- **color: var(--main-color);** → Muda a cor do ícone para **vermelho (#da190f)** ao passar o Mouse.

```
.header .icons div ion-icon:hover {
  color: var(--main-color);
}
```

Ocultando o Botão do Menu (#menu-btn)

- **display: none;** → Esconde o botão do menu, que pode ser ativado com **JavaScript** em telas menores.

```
#menu-btn {
  display: none;
}
```

Propriedade	Função
display: flex;	Organiza os ícones horizontalmente.
align-items: center;	Centraliza os ícones verticalmente.
margin-left: 2rem;	Adiciona espaçamento entre os ícones.
color: #fff;	Define os ícones como brancos.
cursor: pointer;	Transforma os ícones em elementos clicáveis.
font-size: 2.5rem;	Aumenta o tamanho dos ícones.
color: var(--main-color);	Muda a cor do ícone ao passar o mouse.
display: none;	Esconde o botão do menu.

Estilização Geral da Seção home

- **position: relative;** → Permite posicionar elementos filhos com absolute.
- **min-height: 85vh;** → Define altura mínima de **85% da altura da tela**.
- **display: flex;** → Usa **Flexbox** para alinhar o conteúdo.
- **align-items: center;** → Centraliza o conteúdo verticalmente.
- **background: url("./images/home.jpg) no-repeat;** → Define uma imagem de fundo.
- **background-size: cover;** → Faz a imagem cobrir toda a área.
- **background-position: center;** → Centraliza a imagem.
- **border-bottom: var(--border);** → Adiciona uma borda inferior.

```
.home {  
  position: relative;  
  min-height: 85vh;  
  display: flex;  
  align-items: center;  
  background: url("./images/home.jpg) no-repeat;  
  background-size: cover;  
  background-position: center;  
  border-bottom: var(--border);  
}
```

Criando um Efeito de Sobreposição Escura

- **content: "";** → Cria um **pseudo-elemento** vazio.
- **position: absolute;** → Posiciona sobre a imagem de fundo.
- **top: 0; left: 0; width: 100%; height: 100%;** → Ocupa toda a seção.
- **background: rgba(0,0,0, 0.5);** → Adiciona uma sobreposição **semi-transparente** preta.
- **z-index: 1;** → Mantém a camada atrás do texto (z-index: 2 no conteúdo).

```
.home::before {  
    content: "";  
    position: absolute;  
    top: 0;  
    left: 0;  
    width: 100%;  
    height: 100%;  
    background: rgba(0,0,0, 0.5);  
    z-index: 1;  
}
```

Estilização do Conteúdo da home

- **max-width: 60rem;** → Define um **limite de largura** para o texto.
- **position: relative;** → Mantém o conteúdo acima da sobreposição.
- **z-index: 2;** → Garante que o texto fique visível acima da sombra escura.

```
.home .content {  
    max-width: 60rem;  
    position: relative;  
    z-index: 2;  
}
```

Estilização do Título (h3).

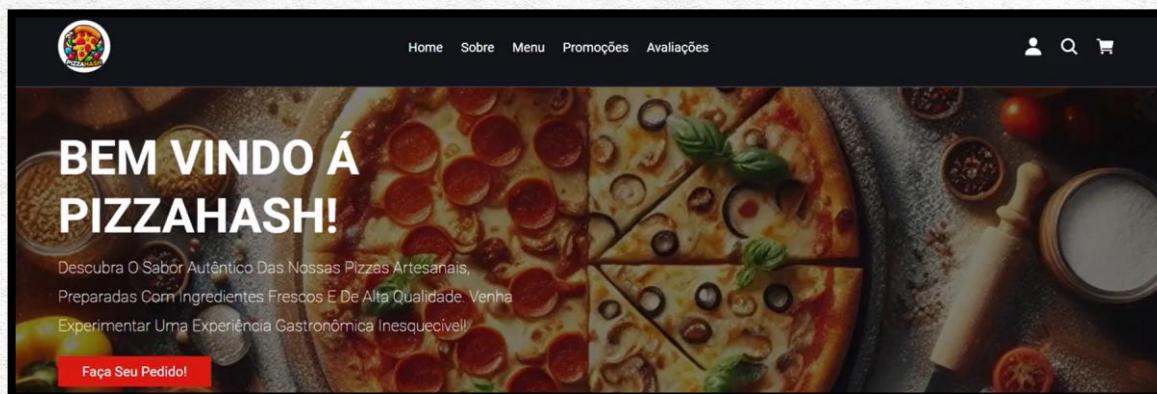
- **font-size: 6rem;** → Deixa o texto grande e chamativo.
- **text-transform: uppercase;** → Transforma o texto em **maiúsculas**.
- **color: #fff;** → Define a cor branca para melhor contraste.

```
.home .content h3 {  
    font-size: 6rem;  
    text-transform: uppercase;  
    color: #fff;  
}
```

Estilização do Parágrafo (p)

- **font-size: 2rem;** → Ajusta o tamanho do texto.
- **font-weight: lighter;** → Deixa o texto mais fino.
- **line-height: 1.8;** → Aumenta o espaçamento entre linhas.
- **padding: 1rem 0;** → Adiciona espaçamento vertical.
- **color: #fff;** → Mantém a cor branca.

```
.home .content p {  
    font-size: 2rem;  
    font-weight: lighter;  
    line-height: 1.8;  
    padding: 1rem 0;  
    color: #fff;  
}
```



Estilização da Seção Sobre

- **display: flex;** → Utiliza **Flexbox** para alinhar a imagem e o texto na mesma linha.
- **align-items: center;** → Mantém os itens centralizados verticalmente.
- **background: var(--bg-card);** → Usa a cor de fundo definida na variável --bg-card.
- **flex-wrap: wrap;** → Permite que os itens se ajustem em telas menores.
- **border: var(--border);** → Adiciona uma borda ao redor da seção.

```
.about .row {  
    display: flex;  
    align-items: center;  
    background: var(--bg-card);  
    flex-wrap: wrap;  
    border: var(--border);  
}
```

Estilização da Imagem

- **flex: 1 1 45rem;** → Define que a imagem ocupará um espaço flexível de **45rem**.
- **width: 100%;** → Faz a imagem se ajustar ao container.

```
.about .row .image {  
    flex: 1 1 45rem;  
}
```

```
.about .row .image img {  
    width: 100%;  
}
```

Estilização do Conteúdo

- **flex: 1 1 45rem;** → Define que o texto também terá um tamanho **flexível** de 45rem.
- **padding: 2rem;** → Adiciona um espaçamento interno para afastar o texto das bordas.

```
.about .row .content {  
    flex: 1 1 45rem;  
    padding: 2rem;  
}
```



Estilização do Título (h3)

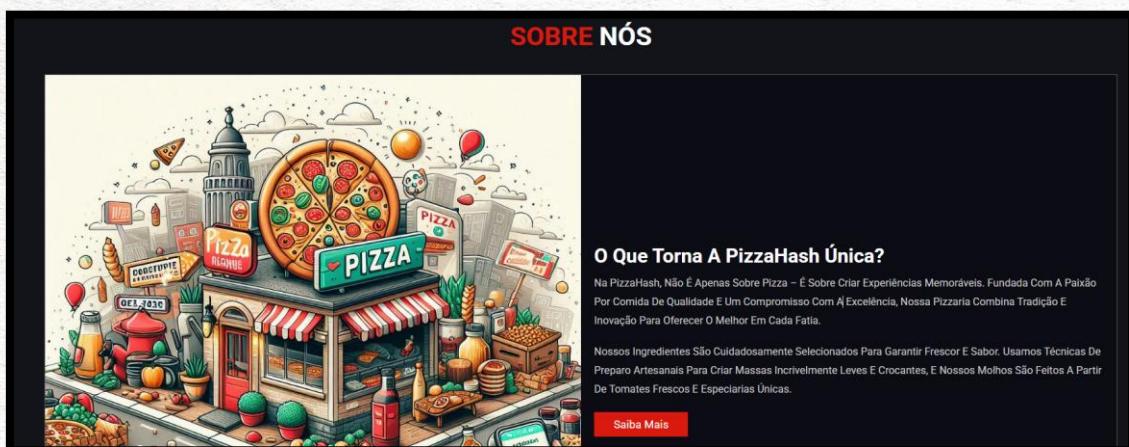
- **font-size: 3rem;** → Ajusta o tamanho do título.
- **color: #fff;** → Mantém a cor branca para contraste.

```
.about .row .content h3 {  
    font-size: 3rem;  
    color: #fff;  
}
```

Estilização do Parágrafo (p)

- **font-size: 1.6rem;** → Define um tamanho legível para o texto.
- **color: #ccc;** → Usa um cinza mais claro para suavizar a leitura.
- **padding: 1rem 0;** → Adiciona espaçamento vertical.
- **line-height: 1.8;** → Aumenta o espaçamento entre linhas para melhor legibilidade.

```
.about .row .content p {  
    font-size: 1.6rem;  
    color: #ccc;  
    padding: 1rem 0;  
    line-height: 1.8;  
}
```



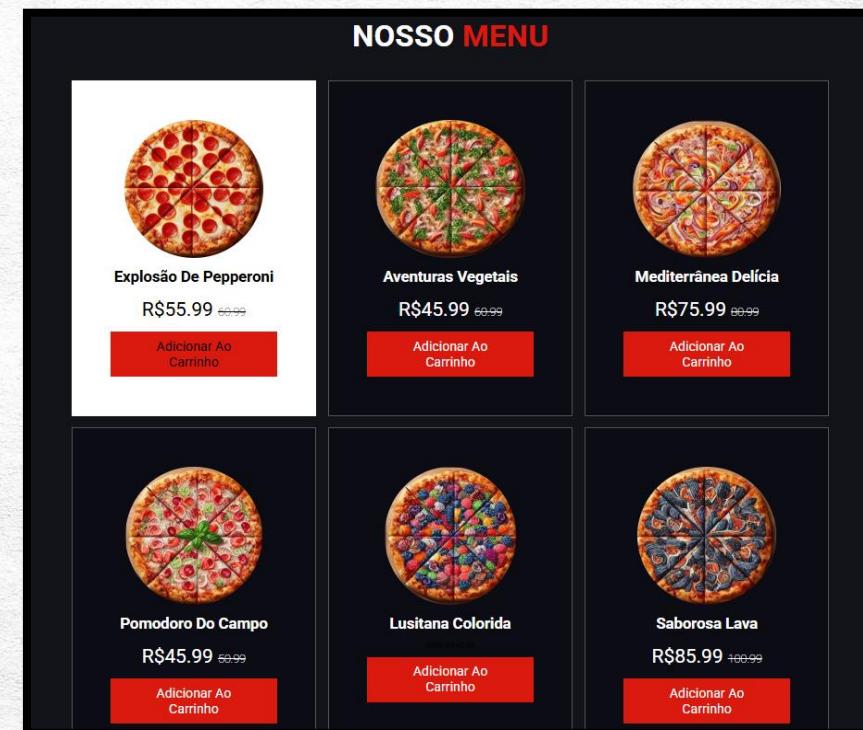
Propriedade	Função
display: flex;	Organiza a imagem e o texto lado a lado.
align-items: center;	Centraliza os elementos verticalmente.
flex-wrap: wrap;	Permite que os elementos se ajustem em telas menores.
background: var(--bg-card);	Define um fundo escuro para destaque.
border: var(--border);	Adiciona uma borda ao redor da seção.
width: 100%;	Garante que a imagem se ajuste ao container.
line-height: 1.8;	Melhora a legibilidade do texto.

Nesta seção, vamos construir a estrutura da página de menu do nosso projeto. O objetivo é exibir uma lista de produtos (neste caso, pizzas) de forma organizada, com imagem, nome, preço e um botão para adicionar ao carrinho.

Estrutura HTML da Página de Menu

A estrutura do menu será criada utilizando a tag <section>, que conterá um título e um contêiner para os itens do menu. Cada item será representado por uma div.box, contendo:

- Uma imagem ilustrativa da pizza;
- O nome da pizza;
- O preço original (tachado) e o preço promocional;
- Um botão para adicionar ao carrinho.



Explicação do Código

Estrutura Principal

- A `<section>` recebe a classe menu e o ID menu, permitindo estilização e navegação.
- O título `<h1>` possui a classe heading, e a palavra "menu" está envolta em ``, permitindo personalização específica via CSS.

Contêiner dos Itens

- O `<div class="box-container">` agrupa todos os itens do menu.
- Cada item do menu está dentro de uma `<div class="box">`, tornando mais fácil a manipulação dos elementos via CSS.

Elementos do Item do Menu

- A imagem do produto é representada por ``, sempre com um alt descritivo para acessibilidade.
- O nome do produto está em `<h3>`.
- O preço atual está dentro de `<div class="price">`, e o preço original aparece dentro de ``, o que facilita a estilização para mostrar que foi reduzido.
- O botão `Adicionar ao carrinho` permite futuras interações.

```
<!-- MENU -->
<section class="menu" id="menu">
  <h1 class="heading">Nosso <span>menu</span> </h1>

  <div class="box-container">
    <div class="box">
      
      <h3>Explosão de Pepperoni</h3>
      <div class="price"> R$55.99 <span>60.99</span></div>
      <a href="#" class="btn">Adicionar ao carrinho</a>
    </div>

    <div class="box">
      
      <h3>Aventuras Vegetais</h3>
      <div class="price"> R$45.99 <span>60.99</span></div>
      <a href="#" class="btn">Adicionar ao carrinho</a>
    </div>

    <div class="box">
      
      <h3>Mediterrânea Delícia</h3>
      <div class="price"> R$75.99 <span>80.99</span></div>
      <a href="#" class="btn">Adicionar ao carrinho</a>
    </div>
  </div>
</section>
```

E vamos aplicar a mesma estrutura, alterando os dados de cada card de pizza.

```
<div class="box">
  
  <h3>Mediterrânea Delícia</h3>
  <div class="price"> R$75.99 <span>80.99</span></div>
  <a href="#" class="btn">Adicionar ao carrinho</a>
</div>

<div class="box">
  
  <h3>Pomodoro do Campo</h3>
  <div class="price"> R$45.99 <span>50.99</span></div>
  <a href="#" class="btn">Adicionar ao carrinho</a>
</div>

<div class="box">
  
  <h3>Lusitana Colorida</h3>
  <div> R$55.99 <span>60.99</span></div>
  <a href="#" class="btn">Adicionar ao carrinho</a>
</div>

<div class="box">
  
  <h3>Saborosa Lava</h3>
  <div class="price"> R$85.99 <span>100.99</span></div>
  <a href="#" class="btn">Adicionar ao carrinho</a>
</div>
</div>
</section>
```



Conclusão

Com essa estrutura, temos um menu funcional e bem organizado. No próximo passo, aplicaremos estilos CSS para tornar a página visualmente atraente e responsiva.

A seção de **Combos e Promoções** é responsável por exibir os produtos em destaque da loja. Utilizamos **CSS Grid** para organizar os elementos de maneira responsiva e garantir um layout moderno e acessível.

Estrutura da Seção

A seção conta com um **container principal** que abriga os produtos. Cada produto é representado por uma "box" que contém:

- Um conjunto de ícones interativos (como adicionar ao carrinho ou favoritar);
- Uma imagem do produto;
- Informações como nome e preço.

Container principal

```
.products .box-container {  
    display: grid;  
    grid-template-columns: repeat(auto-fit, minmax(30rem, 1fr));  
    gap: 1.5rem;  
}
```

Explicação:

- **display: grid;** - Define a disposição dos elementos como um grid;
- **grid-template-columns: repeat(auto-fit, minmax(30rem, 1fr));** - Cria colunas responsivas que se ajustam automaticamente ao tamanho da tela;
- **gap: 1.5rem;** - Define um espaçamento entre os itens.

Estilização dos produtos

Cada produto é envolvido por uma caixa com borda e espaçamento definidos.

- **text-align: center;** - Centraliza os elementos dentro da caixa;
- **border: var(--border);** - Adiciona uma borda padronizada;
- **padding: 2rem;** - Adiciona espaçamento interno.

```
.products .box-container .box {  
    text-align: center;  
    border: var(--border);  
    padding: 2rem;  
}
```

Ícones interativos

Os ícones oferecem funcionalidades interativas, como adicionar ao carrinho.

```
.products .box-container .box .icons a ion-icon {  
    height: 3rem;  
    width: 3rem;  
    line-height: 3rem;  
    font-size: 1.7rem;  
    border: var(--border);  
    color: #fff;  
    margin: 0.3rem;  
}
```

- **height e width: 3rem;** - Define o tamanho do ícone;
- **font-size: 1.7rem;** - Ajusta o tamanho da fonte;
- **border: var(--border);** - Mantém a identidade visual do site;
- **color: #fff;** - Define a cor dos ícones;
- **margin: 0.3rem;** - Garante espaçamento entre os ícones.

Efeito ao passar o mouse:

Altera a cor do fundo ao passar o mouse para indicar interatividade.

```
.products .box-container .box .icons a ion-icon:hover {  
    background: var(--main-color);  
}
```

Estilização da imagem do produto

As imagens precisam ser responsivas e bem posicionadas.

- **padding: 2.5rem 0;** - Adiciona espaçamento superior e inferior;
- **height: 25rem;** - Define uma altura padrão para uniformizar os produtos.

```
.products .box-container .box .image {  
    padding: 2.5rem 0;  
}  
.products .box-container .box .image img {  
    height: 25rem;  
}
```

Nesta parte, continuamos a estilização dos elementos internos da seção de combos e promoções, focando no conteúdo textual e na apresentação dos preços.

1. Títulos dos produtos

- Define a cor do texto como branco (#fff) para destacar sobre o fundo escuro.
- Define o tamanho da fonte como 2.5rem para manter a legibilidade.

```
.products .box-container .box .content h3 {  
    color: #fff;  
    font-size: 2.5rem;  
}
```

2. Estrela de avaliação

- Adiciona um espaçamento interno (“padding”) de 1.5rem entre as estrelas e os demais elementos.
- Define o tamanho das estrelas para 1.9rem.
- Aplica a cor principal da marca (“var(--main-color)”) nas estrelas para manter a identidade visual.

```
.products .box-container .box .content .stars {  
    padding: 1.5rem;  
}  
  
.products .box-container .box .content .stars i {  
    font-size: 1.9rem;  
    color: var(--main-color);  
}
```



3. Estilização do preço

- Define o preço principal na cor branca para maior contraste com o fundo.
- Ajusta o tamanho da fonte para 2.5rem, garantindo destaque ao valor do produto.
- O valor anterior, quando houver desconto, é exibido com uma fonte menor (1.5rem) e peso mais leve para diferenciar do preço principal.
- Usa text-transform: lowercase; para padronizar a apresentação do valor com um estilo mais moderno e limpo.

Com isso, a seção de produtos/promoções está totalmente estilizada, garantindo uma apresentação atraente e bem organizada para os usuários.

```
.products .box-container .box .content .price {  
    color: #fff;  
    font-size: 2.5rem;  
}  
  
.products .box-container .box .content .price span {  
    font-weight: lighter;  
    font-size: 1.5rem;  
    text-transform: lowercase;  
}
```

Nesta aula, vamos estilizar a seção de avaliações do site, garantindo que os depoimentos dos usuários fiquem bem apresentados e harmonizados com o restante do layout.

Estrutura CSS:

1. Container Principal

- Utilizamos **Grid Layout** para organizar os elementos das avaliações.
- O grid-template-columns ajusta automaticamente o tamanho das colunas, garantindo responsividade.
- O gap adiciona espaçamento entre os itens.

```
.review .box-container {  
    display: grid;  
    grid-template-columns: repeat(auto-fit, minmax(30rem, 1fr));  
    gap: 1.5rem;  
}
```

2. Box de Avaliação

- Aplica uma borda conforme a variável --border definida no estilo global.
- Centraliza todo o conteúdo dentro do box.
- Adiciona espaçamento interno (padding) para melhor distribuição dos elementos.

```
.review .box-container .box {  
    border: var(--border);  
    text-align: center;  
    padding: 3rem 2rem;  
}
```



3. Texto da Avaliação

- Define um tamanho de fonte adequado para a leitura.
- Usa um line-height maior para melhor espaçamento entre as linhas.
- Adiciona uma cor suave para contrastar com o fundo escuro.
- Inclui espaçamento interno para dar mais respiro ao texto.

```
.review .box-container .box p {  
    font-size: 1.7rem;  
    line-height: 1.8;  
    color: #ccc;  
    padding: 2rem 0;  
}
```

4. Imagem do Usuário

- Define o tamanho da imagem do usuário.
- Aplica border-radius: 50% para torná-la circular.
- Usa object-fit: cover; para garantir que a imagem preencha corretamente o espaço sem distorção.

```
.user {  
    height: 10rem;  
    width: 10rem;  
    border-radius: 50%;  
    object-fit: cover;  
}
```



5. Nome do Usuário

- Adiciona espaçamento entre os elementos.
- Usa cor branca para destaque.
- Define um tamanho de fonte adequado para visibilidade.

```
.review .box-container .box h3 {  
    padding: 1rem 0;  
    color: #fff;  
    font-size: 2.2rem;  
}
```

6. Estilização das Estrelas de Avaliação

- Define um tamanho de fonte maior para as estrelas.
- Aplica a cor principal do site, destacando a avaliação visualmente.

```
.review .box-container .box .stars i {  
    font-size: 1.7rem;  
    color: var(--main-color);  
}
```

Resumo da Aula:

- Utilizamos Grid Layout para estruturar os depoimentos.
- Aplicamos estilos nas caixas de avaliação, garantindo organização e legibilidade.
- Estilizamos a imagem do usuário para manter um padrão visual circular.
- Ajustamos cores e tamanhos de texto para uma melhor experiência do usuário.
- Destacamos as estrelas de avaliação com a cor principal do site.

Essa estilização garante que a seção de avaliações fique visualmente atraente e responsiva!

Nesta aula, vamos aprender a estilizar o rodapé do nosso projeto. O rodapé é uma seção fundamental de um site, pois contém informações adicionais, links úteis e pode contribuir para a identidade visual da página.

1. Estilizando o título do rodapé

O rodapé inicia com um título estilizado para dar destaque ao seu conteúdo.

```
.heading-footer {  
    text-align: center; /* Centraliza o título */  
    text-transform: uppercase; /* Transforma o texto em maiúsculas */  
    font-size: 2rem; /* Define o tamanho da fonte */  
    color: #fff; /* Define a cor branca para o texto */  
    margin-bottom: 1.5rem; /* Adiciona espaçamento inferior */  
}  
  
.heading-footer span {  
    color: var(--main-color); /* Define a cor principal para o span dentro do título */  
    text-transform: uppercase;  
}
```

2. Estilizando o container principal do rodapé

O rodapé terá um fundo personalizado e um layout flexível para melhor organização dos elementos.

```
.footer {  
    background: var(--bg); /* Define a cor de fundo do rodapé */  
    text-align: center; /* Centraliza o texto */  
    padding: 4rem 0; /* Adiciona espaçamento interno */  
}
```

3. Criando a estrutura flexível do rodapé

Usamos o display: flex; para organizar os elementos em colunas responsivas.

```
.footer .container {  
    display: flex; /* Ativa o flexbox */  
    flex-wrap: wrap; /* Permite que os itens quebrem para a próxima linha se necessário */  
    justify-content: space-between; /* Distribui os itens igualmente pelo espaço disponível */  
    padding: 0 7%; /* Define espaçamento lateral */  
}
```

4. Criando colunas para organização do conteúdo

Cada coluna dentro do rodapé terá um espaçamento adequado para melhor distribuição das informações.

```
.footer-col {  
    flex: 1 1 25rem; /* Define o tamanho flexível da coluna */  
    margin: 1rem; /* Adiciona espaçamento entre as colunas */  
}  
  
.footer-col ul {  
    list-style: none; /* Remove os marcadores da lista */  
    padding: 0; /* Remove espaçamento interno */  
}
```

Com essas estilizações, criamos um rodapé visualmente organizado e responsivo. Na próxima aula, continuaremos aprimorando essa seção.

Nesta segunda parte da estilização do rodapé, vamos adicionar estilos para os links dentro das listas, os ícones das redes sociais e a seção inferior do rodapé.

Estilização dos Links no Rodapé

Os links dentro da lista do rodapé serão estilizados para garantir melhor visualização e interação:

```
/* Estiliza os itens da lista dentro de cada coluna do rodapé */
.footer-col ul li {
    margin-bottom: 1rem; /* Adiciona um espaçamento entre os itens da lista */
}

/* Estiliza os links dentro das listas do rodapé */
.footer-col ul li a {
    color: #ccc; /* Define a cor dos links */
    text-decoration: none; /* Remove o sublinhado padrão dos links */
    font-size: 1.6rem; /* Ajusta o tamanho da fonte */
    transition: color 0.3s; /* Adiciona uma transição suave na mudança de cor */
}

/* Muda a cor dos links ao passar o mouse sobre eles */
.footer-col ul li a:hover {
    color: var(--main-color); /* Define a cor ao passar o mouse */
```



Estilização das Redes Sociais

Agora, vamos definir o estilo dos ícones de redes sociais para que fiquem centralizados e com efeito ao passar o mouse:

```
/* Estiliza os ícones das redes sociais no rodapé */
.footer .social-links {
    display: flex;
    gap: 1rem; /* Define o espaçamento entre os ícones */
    justify-content: center; /* Centraliza os ícones na seção */
    margin-top: 1.5rem; /* Adiciona espaçamento acima dos ícones */
}

/* Estiliza os links das redes sociais */
.footer .social-links a {
    color: #ccc; /* Define a cor dos ícones das redes sociais */
    font-size: 3rem; /* Ajusta o tamanho dos ícones */
    transition: color 0.3s; /* Adiciona uma transição suave ao mudar de cor */
}

/* Muda a cor dos ícones das redes sociais ao passar o mouse */
.footer .social-links a:hover {
    color: var(--main-color); /* Define a cor ao passar o mouse */
}
```

Estilização da Parte Inferior do Rodapé

A parte inferior do rodapé (footer-bottom) geralmente contém direitos autorais ou informações adicionais:

```
/* Estiliza a parte inferior do rodapé */
.footer-bottom {
    text-align: center; /* Centraliza o texto */
    padding: 1rem 0; /* Adiciona espaçamento interno */
    font-size: 1.6rem; /* Ajusta o tamanho do texto */
    border-top: var(--border); /* Adiciona uma borda no topo para separar do restante */
    margin-top: 2rem; /* Adiciona espaçamento acima da borda */
    color: #ccc; /* Define a cor do texto */
}
```

Com isso, finalizamos a estilização do rodapé! Se precisar de ajustes ou dúvidas, é só avisar! 

Nesta aula, vamos aprender a tornar nossa landing page responsiva utilizando **Media Queries** no CSS. Isso garante que nosso site se adapte bem a diferentes tamanhos de tela, como desktops, tablets e dispositivos móveis.

O que são Media Queries?

Media Queries são regras do CSS que permitem aplicar estilos diferentes dependendo do tamanho da tela ou de outras condições do dispositivo. A sintaxe básica de uma Media Query é:

```
@media (condição) {  
    /* Estilos que serão aplicados se a condição for verdadeira */  
}
```

Agora, vamos aplicar Media Queries para ajustar os elementos da nossa landing page.

Ajustes para telas menores que 991px

Abaixo de 991px, fazemos ajustes gerais para que os elementos fiquem mais proporcionais em telas menores:

```
@media (max-width: 991px) {  
    html {  
        font-size: 55%; /* Reduz o tamanho da fonte global */  
    }  
  
    .header {  
        padding: 1.5rem 2rem; /* Ajusta o espaçamento do cabeçalho */  
    }  
  
    section {  
        padding: 2rem; /* Reduz o espaçamento das seções */  
    }  
}
```

Ajustes para telas menores que 768px

Quando a tela for menor que 768px, aplicamos mudanças mais significativas, como transformar o menu em um botão:

```
@media (max-width: 768px) {  
    #menu-btn {  
        display: inline-block; /* Exibe o botão do menu */  
    }  
  
.header .navbar {  
    position: absolute;  
    top: 100%;  
    right: -100%;  
    background: #fff;  
    width: 30rem;  
    height: calc(100vh - 9.5rem); /* Ajusta a altura do menu */  
}  
  
.header .navbar a {  
    color: var(--bg-card);  
    display: block;  
    margin: 1.5rem;  
    padding: 0.5rem;  
    font-size: 2rem; /* Ajusta o tamanho dos links do menu */  
}  
}
```



Ajustes para a Home

Para a seção **Home**, ajustamos o alinhamento do texto e o tamanho da fonte:

```
@media (max-width: 768px) {  
    .home {  
        background-position: left;  
        justify-content: center;  
        text-align: center;  
    }  
  
    .home .content h3 {  
        font-size: 4.5rem; /* Reduz o tamanho do título */  
    }  
  
    .home .content p {  
        font-size: 1.5rem; /* Reduz o tamanho do parágrafo */  
    }  
}
```

Conclusão

Com essas regras, nossa landing page já se adapta melhor a dispositivos menores. Podemos continuar refinando a responsividade adicionando mais Media Queries para diferentes tamanhos de tela. Nos próximos passos, vamos aprimorar a navegação mobile e testar nossa página em diferentes dispositivos!

Isso conclui a nossa aula sobre responsividade. Pratique ajustando diferentes elementos do seu site para garantir uma boa experiência do usuário em qualquer dispositivo!



Módulo 9 – INTRODUÇÃO AO FRONT-END

DOM, FORMULÁRIOS E LOCALSTORAGE

DOM, FORMULÁRIOS E LOCALSTORAGE

DOM, FORMULÁRIOS E LOCALSTORAGE



Nos dias atuais, a importância do front end em um site, impulsionada especialmente pelo uso de tecnologias como JavaScript, transcendeu o mero aspecto estético para se tornar uma peça central na entrega de experiências digitais excepcionais. O front end não é apenas a face visível de um site, mas sim a interface através da qual os usuários interagem e exploram conteúdos, serviços e funcionalidades.

No âmbito dessa interação, o JavaScript emerge como um componente essencial, conferindo dinamismo, interatividade e funcionalidade aos elementos front end. Essa linguagem de programação possibilita a criação de interfaces fluidas e responsivas, enriquecendo a experiência do usuário e elevando a usabilidade a patamares mais elevados.

Ao explorar a importância do front end, é crucial reconhecer que a primeira impressão de um site muitas vezes determina a permanência do usuário. Uma interface bem projetada, com animações suaves, transições elegantes e interações intuitivas, cativa o visitante e o incentiva a explorar mais profundamente o conteúdo oferecido.

Além disso, o JavaScript desempenha um papel vital na otimização do desempenho do site, permitindo a implementação de técnicas como carregamento assíncrono de conteúdo e pré-processamento de recursos, o que resulta em tempos de carregamento mais rápidos e uma experiência de navegação mais fluida.

A importância do front end, com ênfase em tecnologias como JavaScript, transcende a simples estética visual. É a chave para criar experiências digitais envolventes, funcionais e memoráveis, que não apenas atendem às expectativas dos usuários, mas as superam, estabelecendo um vínculo duradouro entre o público e o site.

A interatividade e o dinamismo são elementos fundamentais para a experiência do usuário em um site, e o JavaScript desempenha um papel central na criação desses recursos no front end. Por meio dessa linguagem de programação, os desenvolvedores podem adicionar uma camada de interatividade que transforma um site estático em uma plataforma dinâmica e envolvente.

A interatividade proporcionada pelo JavaScript permite que os usuários realizem ações como clicar, arrastar, digitar e rolar em uma página da web, gerando respostas imediatas e visíveis. Essa capacidade de resposta instantânea cria uma sensação de fluidez e controle, mantendo os usuários engajados e incentivando a exploração mais profunda do conteúdo.

Além disso, o dinamismo possibilitado pelo JavaScript permite a atualização contínua e assíncrona do conteúdo, sem a necessidade de recarregar toda a página. Isso é particularmente evidente em recursos como feeds de mídia social em tempo real, sistemas de chat e atualizações automáticas de dados. Essa abordagem de carregamento dinâmico não apenas melhora a eficiência do site, mas também proporciona uma experiência mais fluida e contínua para o usuário.

Outro aspecto importante é a capacidade do JavaScript de criar animações e transições suaves, que adicionam um elemento visualmente atraente ao site. Seja ao navegar por uma galeria de imagens, assistir a uma apresentação de slides ou simplesmente rolar pela página, essas animações dinâmicas tornam a experiência de navegação mais envolvente e cativante.

Além disso, o JavaScript desempenha um papel fundamental na garantia da responsividade do site, permitindo que ele se adapte de forma dinâmica e fluida a diferentes dispositivos e tamanhos de tela. Isso assegura que a experiência do usuário seja consistente e agradável, independentemente do dispositivo utilizado para acessar o site.

Nesse momento vamos fazer uma relação do Front-end com um Quebra-cabeça para entender cada peça que forma a interface visual de um site.

- **Peças do Quebra-cabeça = Tecnologias do Front-end**

As peças do quebra-cabeça do desenvolvimento Front-End são essenciais para criar a experiência visual e interativa de um site ou aplicativo. Assim como no quebra-cabeça, cada peça desempenha um papel único e se encaixa perfeitamente com as outras para formar um todo coeso.

- **HTML (Hypertext Markup Language):** Imagine o HTML como as peças de borda do quebra-cabeça. Ele fornece a estrutura básica e o esqueleto do seu projeto, definindo os elementos como cabeçalhos, parágrafos, imagens e links. Cada peça HTML define uma parte específica da página, como uma imagem ou um título, que se encaixa de forma organizada para criar a estrutura visual do seu projeto.

- **CSS (Cascading Style Sheets):** O CSS é como as peças coloridas do quebra-cabeça que dão vida à imagem. Ele controla a apresentação e o estilo do conteúdo HTML, permitindo que você defina cores, fontes, espaçamentos e layouts. Assim como as peças de cores diferentes dão forma e estilo ao quebra-cabeça, o CSS transforma a estrutura básica do HTML em um design visualmente atraente e coerente.

- **JavaScript:** O JavaScript é a peça dinâmica do quebra-cabeça. Ele adiciona interatividade e funcionalidade à sua página, permitindo que você crie animações, responda a eventos do usuário e manipule o conteúdo em tempo real. Assim como as peças que se movem e se encaixam em diferentes lugares do quebra-cabeça, o JavaScript permite que sua página se adapte e responda às ações dos usuários, tornando a experiência mais envolvente e interativa.

Quando você domina essas peças do quebra-cabeça do Front-End - HTML, CSS e JavaScript - você pode criar projetos web impressionantes e funcionais, onde cada elemento se encaixa harmoniosamente para proporcionar uma experiência de usuário excepcional.



2. Resolução de problemas = Encaixando as peças

Resolver problemas de programação é como montar um quebra-cabeça, e o JavaScript é a ferramenta que nos permite encaixar as peças de forma dinâmica e funcional. Assim como um quebra-cabeça desafia a mente a encontrar soluções para encaixar as peças corretamente, a resolução de problemas em JavaScript exige criatividade, lógica e habilidades de pensamento crítico.



- **Identificar as peças:** Assim como em um quebra-cabeça, o primeiro passo para resolver um problema em JavaScript é identificar as peças disponíveis. Isso significa compreender os elementos do problema, como os dados de entrada, as condições e as operações necessárias para chegar à solução.
- **Analizar e planejar:** Após identificar as peças, é hora de analisar e planejar como elas se encaixam. Isso envolve entender a lógica por trás do problema e decidir a melhor abordagem para alcançar a solução. É como visualizar a imagem completa do quebra-cabeça antes de começar a montagem.
- **Encaixar as peças:** Com um plano em mente, é hora de começar a encaixar as peças do problema. Isso pode envolver escrever código JavaScript para realizar cálculos, manipular dados, controlar o fluxo do programa e interagir com o usuário. Às vezes, as peças se encaixam perfeitamente de primeira, enquanto em outras situações é necessário tentativa e erro para encontrar a combinação correta.
- **Testar e ajustar:** Assim como montar um quebra-cabeça pode exigir ajustes para que todas as peças se encaixem perfeitamente, resolver problemas em JavaScript requer testes e ajustes contínuos. É importante testar o código para garantir que ele funcione conforme esperado e fazer ajustes conforme necessário para corrigir erros ou melhorar a eficiência.

3. Imagem final = Experiência do Usuário

A imagem final de qualquer projeto em JavaScript é a experiência do usuário. Assim como a conclusão de um quebra-cabeça resulta na imagem completa, a finalização do código JavaScript resulta na experiência interativa e envolvente que o usuário vivencia.

- **Interatividade fluente:** JavaScript permite criar interfaces dinâmicas e interativas. A imagem final é uma experiência em que os elementos da página respondem de forma rápida e suave aos comandos do usuário. Seja ao clicar em um botão, rolar uma página ou preencher um formulário, a interatividade é fluida e responsiva.
- **Feedback imediato:** Uma experiência de usuário eficaz inclui feedback imediato. Isso significa que, ao interagir com os elementos da página, o usuário recebe respostas claras e instantâneas. Seja através de animações, mensagens de confirmação ou atualizações em tempo real, o JavaScript permite fornecer feedback imediato para melhorar a usabilidade e a experiência do usuário.
- **Personalização e adaptação:** Cada usuário é único, e a imagem final da experiência em JavaScript leva em consideração essa diversidade. Por meio de técnicas de personalização e adaptação, como armazenamento de preferências do usuário, detecção de dispositivos e ajustes dinâmicos de layout, a experiência do usuário é otimizada para atender às necessidades individuais de cada pessoa.
- **Fluxo intuitivo:** Uma boa experiência do usuário é aquela em que o fluxo de navegação é intuitivo e fácil de entender. Com JavaScript, é possível criar transições suaves entre páginas, menus que se expandem e contraem conforme necessário e funcionalidades de arrastar e soltar que simplificam a interação do usuário.
- **Performance otimizada:** Por fim, a imagem final da experiência do usuário em JavaScript inclui uma performance otimizada. Isso significa que a aplicação responde rapidamente aos comandos do usuário, mesmo em dispositivos e conexões de internet mais lentos. O código é eficiente e bem otimizado para garantir uma experiência fluida e sem interrupções.

4. Criatividade = Personalização

Podemos combinar essas peças de maneiras infinitas para criar páginas web distintas e personalizadas. A criatividade entra em cena quando exploramos novas maneiras de organizar essas peças, criando layouts inovadores e interações envolventes. A personalização, por sua vez, nos permite adaptar essas experiências para atender às necessidades específicas de cada projeto ou usuário.

Cada projeto é único, com seus próprios requisitos, objetivos e público-alvo. Portanto, a habilidade de personalização é essencial para garantir que cada página web atenda precisamente às demandas e expectativas de quem a utiliza. Isso significa que, ao invés de seguir um modelo padrão, podemos ajustar cada elemento da interface para se adequar ao contexto específico de cada projeto. Por exemplo, um site de e-commerce pode requerer uma personalização que destaque produtos populares, apresentando-os de forma proeminente na página inicial, enquanto um blog pode preferir uma abordagem que priorize os últimos artigos publicados. Da mesma forma, a interface de um aplicativo de saúde pode ser personalizada para exibir informações relevantes de saúde do usuário, enquanto um site de notícias pode optar por destacar as últimas manchetes.

A personalização não se limita apenas ao conteúdo visível, mas também se estende à funcionalidade e usabilidade da página. Botões de ação, formulários de inscrição, opções de navegação - tudo pode ser adaptado para atender às necessidades específicas de cada projeto, garantindo uma experiência otimizada e intuitiva para os usuários.

Portanto, ao desenvolver páginas web, é fundamental não apenas ser criativo na montagem das peças do quebra-cabeça, mas também ser capaz de personalizar cada projeto com a especificidade que ele precisa. Essa abordagem garante que cada página seja verdadeiramente única e capaz de alcançar seus objetivos de forma eficaz, proporcionando uma experiência memorável e relevante para os usuários.

5. Encaixando Peças Especiais = Integração com Serviços e APIs

Assim como em um quebra-cabeça, onde cada peça tem sua função específica, no desenvolvimento web, algumas peças são especiais: as APIs e serviços externos. Integrar essas peças especiais em nossos projetos é como encaixar elementos únicos que trazem funcionalidades adicionais e dados dinâmicos para nossas páginas.

A integração com serviços e APIs nos permite expandir as capacidades de nossos projetos web, aproveitando recursos externos para fornecer funcionalidades avançadas. Podemos incorporar serviços de mapeamento para exibir localizações, APIs de pagamento para facilitar transações seguras, ou até mesmo serviços de análise para acompanhar o desempenho do nosso site.

Essas peças especiais, quando integradas de forma inteligente, podem elevar significativamente a experiência do usuário e a eficácia de nossos projetos. No entanto, assim como no quebra-cabeça, é crucial encontrar o encaixe perfeito e garantir que essas peças se integrem harmoniosamente ao restante do projeto.

Portanto, ao desenvolver páginas web, não devemos apenas nos concentrar em montar o quebra-cabeça, mas também em encaixar essas peças especiais com maestria, aproveitando ao máximo as oportunidades que elas oferecem para criar experiências web verdadeiramente excepcionais.



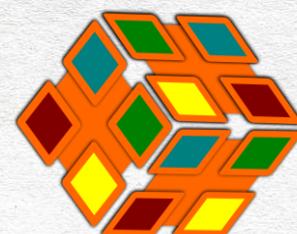
6. Construindo um Quebra-Cabeça Tridimensional = Desenvolvimento de Aplicativos Web e Sistema mais Complexos

Em um quebra-cabeça tridimensional, o desenvolvimento de aplicativos web e sistemas mais complexos exige uma abordagem multifacetada e estruturada. Nesse contexto, os frameworks de JavaScript desempenham um papel fundamental, fornecendo um conjunto robusto de ferramentas e estruturas que facilitam a construção de projetos web sofisticados.

Desenvolver aplicativos web e sistemas complexos é como montar um quebra-cabeça em três dimensões, onde além das peças individuais, também precisamos considerar a profundidade e a interconexão entre os elementos. Os frameworks de JavaScript, como React, Angular e Vue.js, oferecem uma abordagem estruturada para lidar com a complexidade, permitindo dividir nossos projetos em componentes reutilizáveis e gerenciar o estado da aplicação de forma eficiente.

Ao utilizar esses frameworks, podemos criar interfaces de usuário dinâmicas e responsivas, implementar lógica de negócios complexa e interagir com serviços externos de maneira assíncrona. Além disso, eles nos proporcionam ferramentas poderosas para otimizar o desempenho e a escalabilidade de nossos aplicativos, garantindo uma experiência fluida para os usuários, independentemente da complexidade do sistema.

Assim como montamos um quebra-cabeça tridimensional, o desenvolvimento de aplicativos web e sistemas mais complexos requer paciência, habilidade e um entendimento claro das ferramentas e técnicas disponíveis. Os frameworks de JavaScript nos fornecem o conjunto de peças necessário para construir projetos web de última geração, onde cada componente se encaixa perfeitamente para criar uma experiência coesa e poderosa para os usuários.



O que é Front-End?

O Front-End lida com a implementação da interface do usuário, interatividade e experiência visual em um site ou aplicativo.

Linguagens principais:

- HTML (HyperText Markup Language): Responsável pela estrutura básica da página.
- CSS (Cascading Style Sheets): Responsável por adicionar estilo e aparência ao conteúdo HTML.
- JavaScript: Adiciona comportamentos interativos, como botões clicáveis ou animações.

O Papel do Desenvolvedor Front-End:

O Desenvolvedor Front-End é como um construtor que trabalha na "fachada" digital. Ele desenvolve a parte visual e interativa do site, integrando designs fornecidos por Web Designers com código funcional.

Ferramentas Comuns:

- Frameworks como React, Angular ou Vue.js são utilizados para criar interfaces interativas e dinâmicas.

Explorando o Web Design:

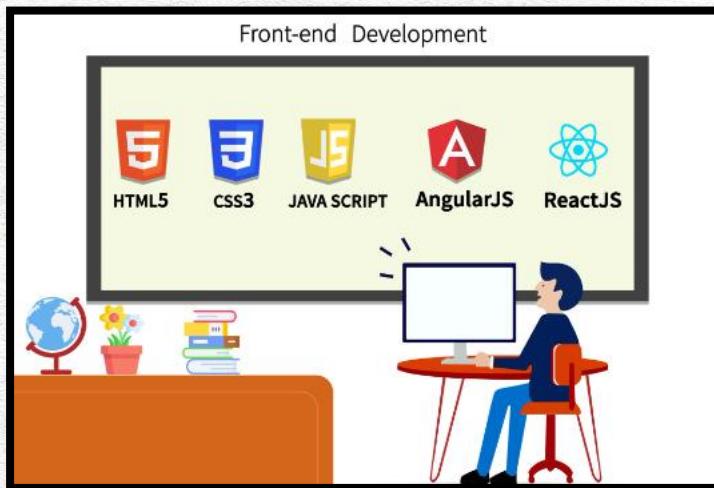
O Web Design envolve a criação visual e estética de um site, incluindo elementos gráficos, cores, layout e tipografia. Ele se concentra na experiência do usuário (UX) e na usabilidade.

O Papel do Web Designer:

O Web Designer é como um arquiteto que projeta a "fachada" digital. Ele cria o conceito visual do site, considerando a identidade da marca e as preferências do público-alvo. Utiliza ferramentas de design gráfico, como Adobe Photoshop, Sketch ou Figma, para materializar suas ideias e colabora com Desenvolvedores Front-End para garantir uma implementação precisa do design.

Colaboração Essencial: Front-End e Web Design:

Ambos os profissionais precisam colaborar estreitamente para garantir que o design seja fielmente implementado e que a experiência do usuário seja excepcional. Eles compartilham responsabilidades na otimização da velocidade do site, acessibilidade e usabilidade, trabalhando juntos para criar uma experiência digital memorável e eficaz.



Este slide oferecerá uma visão geral dos tópicos que exploraremos neste módulo. Desde eventos em HTML até manipulações avançadas do DOM, passando por renderização de páginas web e o poderoso Javascript para eventos dinâmicos. Além disso, vamos desvendar os segredos dos formulários e descobrir como utilizar o LocalStorage para armazenar dados localmente.

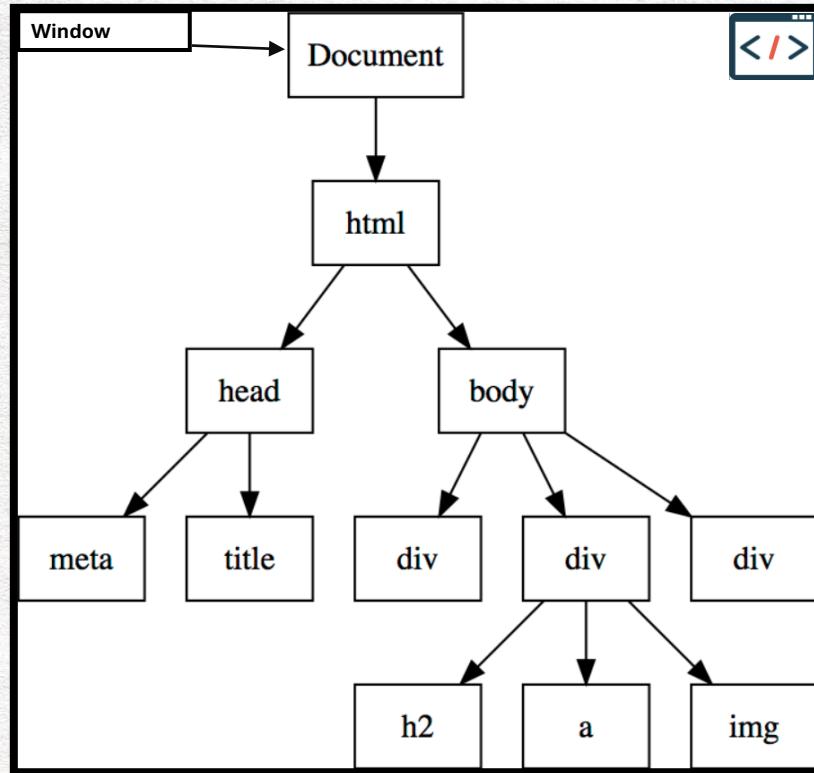
O que vamos abordar nesse módulo:

- Eventos em HTML
- DOM – Document Object Model
- Renderização de uma página web
- Manipulações do DOM
- Eventos com Javascript
- Formulários
- LocalStorage



O que é DOM?

O DOM (Document Object Model) é uma representação hierárquica em forma de árvore de elementos HTML em uma página web, permitindo interações e manipulações dinâmicas por meio de linguagens como JavaScript.



```
index.html > ...
1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="UTF-8" />
5      <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6      <title>Document</title>
7    </head>
8    <body>
9      <div></div>
10     <div></div>
11     <div>
12       <h2></h2>
13       <a href=""></a>
14       <img src="" alt="" />
15     </div>
16   </body>
17 </html>
```

O DOM (Document Object Model) é basicamente uma forma organizada de entender e interagir com uma página da web. Ele funciona como uma árvore, onde cada parte da página (como títulos, textos, imagens, etc.) é representada como um "objeto". Isso permite que você mexa e mude esses objetos usando JavaScript, tornando a página dinâmica e interativa.

- **Document:** Refere-se ao documento HTML, XHTML ou XML sendo processado, representando a janela do navegador ou o conteúdo carregado nele.
- **Object:** Indica que cada elemento (como tags HTML, atributos, texto) dentro do documento é representado como um objeto, e esses objetos são acessíveis e manipuláveis por meio de código.
- **Model:** Refere-se à representação estruturada do documento como uma árvore de objetos interconectados, onde cada nó na árvore representa um elemento individual do documento.

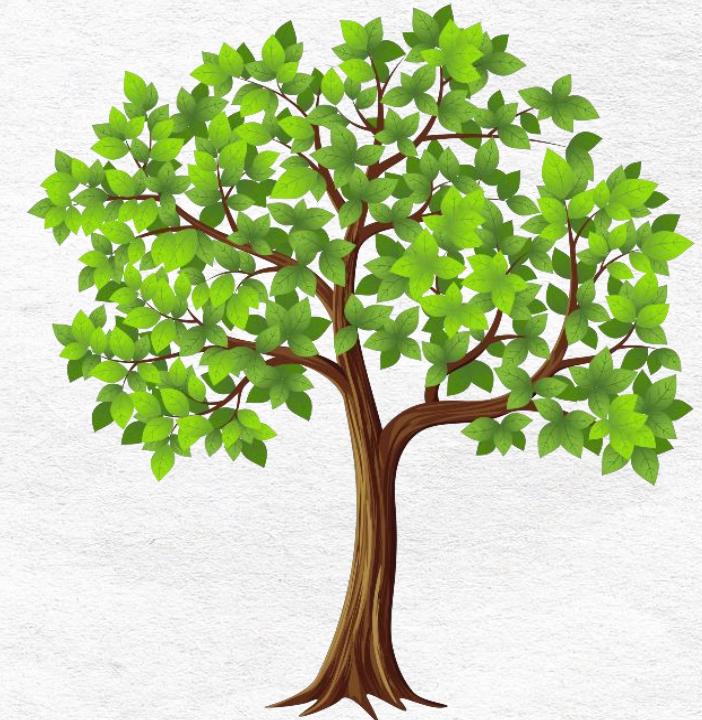
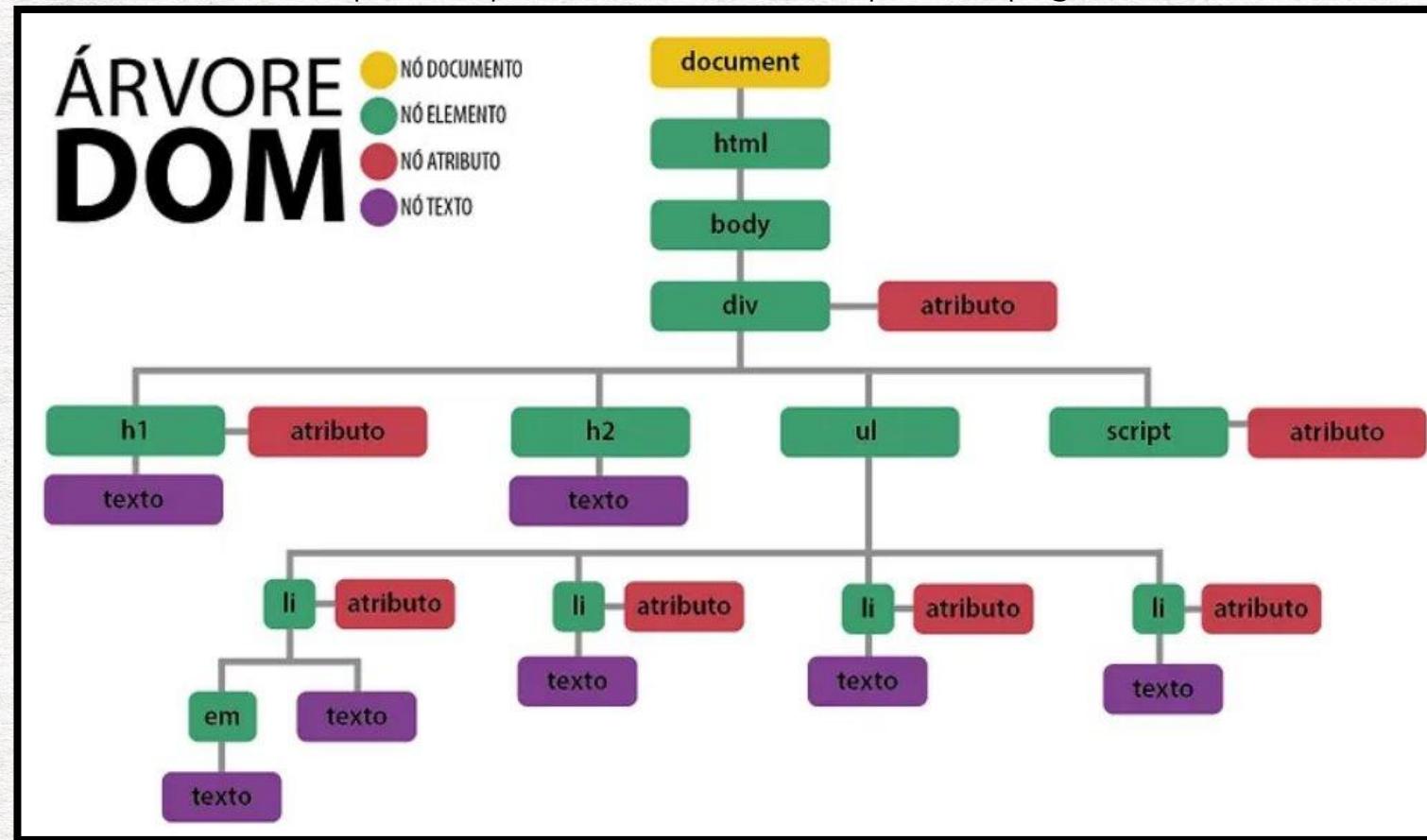
Em resumo, o Document Object Model é uma "modelagem" (representação) do "documento" (página da web) como uma estrutura hierárquica de "objetos" interligados, permitindo a manipulação desses objetos para interações dinâmicas e alterações no conteúdo, estilo e estrutura da página por meio de linguagens de script como JavaScript.

Antes do Document Object Model (DOM), é importante entender o papel do objeto global window no navegador. Este objeto representa a janela do navegador e é fundamental para que o JavaScript interaja com o ambiente do navegador. Serve como o ponto de entrada para várias interações.

O objeto window oferece acesso a uma série de recursos importantes, como métodos para controlar a própria janela, gerenciar frames, e até mesmo controlar o histórico de navegação. Além disso, disponibiliza diversas propriedades e métodos úteis para interagir com a página web, como manipulação de cookies, obtenção de informações sobre o navegador e muito mais.

Nó na árvore DOM

Um nó na árvore do DOM (Document Object Model) é qualquer elemento individual, como um elemento HTML, um atributo, um texto ou um comentário, que compõe a estrutura hierárquica da página web.



No contexto do Document Object Model (DOM), um "nó" ou "node" refere-se a qualquer entidade individual na estrutura hierárquica de um documento HTML. Cada elemento, atributo, texto, comentário ou nó de documento é representado como um nó na árvore do DOM. Esses nós estão conectados entre si, formando uma estrutura em árvore que espelha a estrutura e o conteúdo da página web. Existem vários tipos de nós comuns no DOM, incluindo:

- **Element Nodes (Nós de Elementos):** Representam os elementos HTML, como `<div>`, `<p>`, ``, e outros. Eles podem conter outros tipos de nós, como nós de texto, nós de atributo e outros elementos.
- **Text Nodes (Nós de Texto):** Responsáveis pelo conteúdo textual dentro de um elemento HTML. Por exemplo, o texto dentro de um parágrafo (`<p>Texto aqui</p>`) está contido em um nó de texto.
- **Attribute Nodes (Nós de Atributos):** Representam os atributos de um elemento HTML, como `id`, `class`, `src`, entre outros. Eles estão associados aos nós de elementos.
- **Document Nodes (Nós de Documento):** Representam o documento como um todo. O nó de documento é o nó raiz da árvore do DOM.
- **Comment Nodes (Nós de Comentários):** Representam os comentários dentro do código HTML.

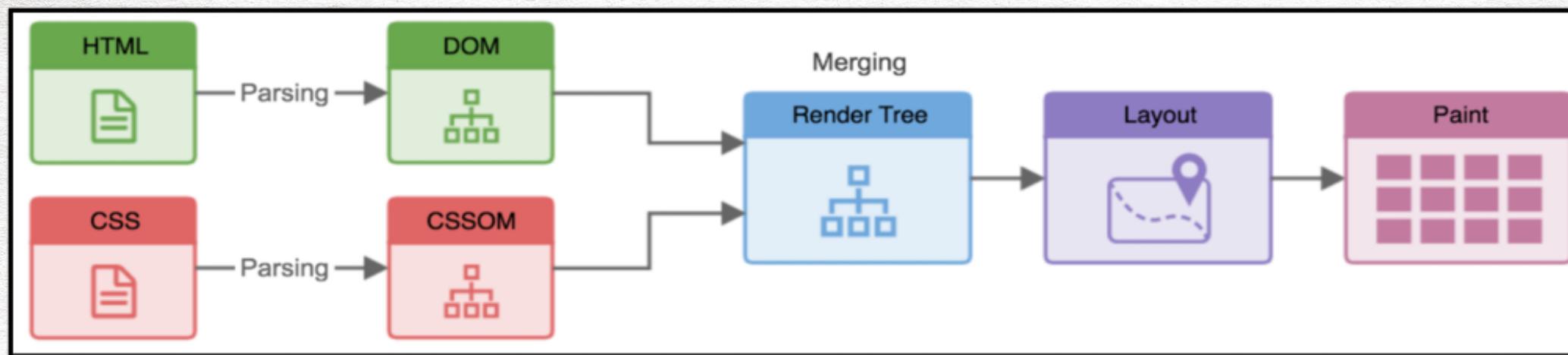
Cada nó no DOM possui propriedades e métodos que podem ser acessados e manipulados por linguagens de script, como JavaScript. Através dessas propriedades e métodos, é possível interagir dinamicamente com os elementos da página, alterando seu conteúdo, estilo, atributos, posição na página, e outros aspectos.

Compreender os nós no DOM é crucial para a manipulação eficiente e a interação dinâmica com os elementos de uma página web, possibilitando a criação de aplicações interativas e dinâmicas.

O que é Renderização?

A renderização é um processo complexo e multietapas que resulta na exibição visual de uma página da web no navegador, permitindo a interação do usuário com o conteúdo apresentado. Esse processo pelo qual um navegador irá interpretar o código HTML, CSS e JavaScript de uma página da web e exibir visualmente o conteúdo para o usuário final. Esse processo envolve várias etapas para que a **Composição** dos elementos sejam organizados em camadas e compostos na tela para formar a visualização final da página. Isso inclui a sobreposição de elementos, transparências e animações.

Agora vamos entender um pouco mais sobre as etapas da renderização:



Etapas da Renderização:

HTML Parsing (Análise HTML): O navegador interpreta o HTML, construindo uma estrutura hierárquica conhecida como o DOM (Document Object Model). Cada elemento HTML é representado como um nó no DOM.

CSS Parsing (Análise CSS): Após a construção do DOM, o navegador analisa o CSS associado à página, criando o CSSOM (CSS Object Model). Isso representa as regras de estilo e como elas se aplicam aos elementos do DOM.

Construção da Árvore de Renderização: O DOM e o CSSOM são combinados para criar a Árvore de Renderização. Essa árvore representa visualmente como os elementos devem ser exibidos, levando em consideração a estrutura do documento e as regras de estilo.

Layout (Reflow): Com a Árvore de Renderização formada, o navegador determina as dimensões e posições exatas de cada elemento na tela. Este é o processo de layout, também conhecido como reflow, que é acionado por mudanças na estrutura do DOM ou nas dimensões dos elementos.

Pintura (Repaint): Finalmente, o navegador pinta os pixels na tela de acordo com as informações da Árvore de Renderização e as dimensões calculadas durante o layout. Este é o processo de pintura, também chamado de repaint, que é acionado por mudanças visuais, como alterações nas cores ou visibilidade.

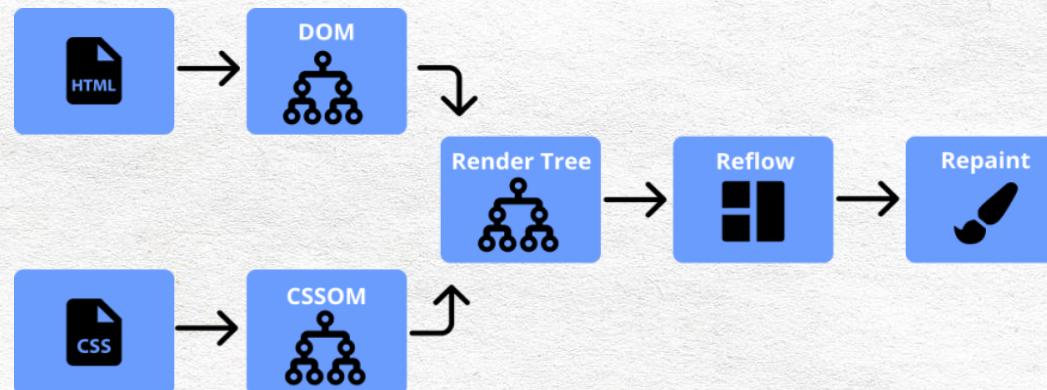
Atualizações Dinâmicas: Se houver interações do usuário ou alterações dinâmicas na página (por meio de JavaScript, por exemplo), o navegador pode repetir alguns desses passos para refletir as mudanças na tela.

Reflow (Layout) e Repaint (Painting): Entendendo os detalhes

Reflow (ou Refluxo de Layout): Durante o reflow, o navegador recalcula as posições e tamanhos dos elementos na página devido a mudanças no DOM ou nos estilos CSS. Isso acontece quando elementos são adicionados, removidos, ou suas propriedades de tamanho, posição ou visibilidade são alteradas. É crucial otimizar o número de reflows para garantir um desempenho eficiente, já que eles consomem recursos computacionais. Estratégias como agrupar mudanças, usar classes de CSS eficientes e evitar manipulações desnecessárias do DOM ajudam a minimizar os reflows.

Repaint (ou Repintura): Durante o repaint, o navegador atualiza visualmente os pixels na tela conforme as mudanças de estilo (cores, bordas, sombras etc.) aplicadas aos elementos. Isso ocorre após o reflow, quando as mudanças de layout são processadas. Minimizar repaints desnecessários é essencial para garantir uma experiência do usuário eficiente, já que repaints consomem recursos de processamento. Estratégias como usar CSS eficiente, evitar mudanças de estilo em grandes áreas da página e utilizar técnicas como transformações 3D podem ajudar a reduzir repaints.

Ao compreender o ciclo de renderização e a importância do reflow e repaint, podemos criar páginas web responsivas e eficientes.



Criando um Arquivo index.html no VS Code e Entendendo a Estrutura Básica

Passo 1: Crie um novo arquivo clicando em **File** (Arquivo) > **New File** (Novo Arquivo). Salve o arquivo com o nome **index.html** na pasta do seu projeto.

Passo 2: A estrutura básica de um documento HTML envolve a definição de tags principais que formam a estrutura do conteúdo da página. Vamos criar uma estrutura HTML simples, utilizando o atalho **! + Enter**

Vamos entender o que cada parte desse código faz:

- **<!DOCTYPE html>**: Declara o tipo de documento HTML.
- **<html lang="pt-BR">**: Define o início do documento HTML e especifica o idioma.
- **<head>**: Contém informações sobre o documento, como metadados, scripts e estilos.
- **<meta charset="UTF-8">**: Define a codificação de caracteres para UTF-8, que suporta a maioria dos caracteres do mundo.
- **<meta name="viewport" content="width=device-width, initial-scale=1.0">**: Controla a escala e o dimensionamento em dispositivos móveis.
- **<title>Minha Página</title>**: Define o título da página exibido na aba do navegador.
- **<body>**: Contém o conteúdo visível da página, como cabeçalho, corpo e rodapé.

A Estrutura de Documento do Modelo de Objeto (DOM) é uma representação em árvore dos elementos HTML em uma página web. Quando você cria um arquivo **index.html** e o abre em um navegador, o navegador constrói uma árvore DOM com base no seu código HTML. Cada tag HTML se torna um nó na árvore DOM.

O Live Server é uma extensão para o VS Code que permite visualizar as alterações em tempo real no seu navegador enquanto você edita o código. Quando você salva seu arquivo **index.html** no VS Code e tem o Live Server ativado, ele automaticamente atualiza a página no navegador para refletir suas alterações.

Isso significa que você pode ver como sua estrutura HTML se traduz em uma árvore DOM e como essas alterações são renderizadas no navegador enquanto você trabalha no código, tornando o processo de desenvolvimento mais eficiente e interativo.

Experimente fazer algumas alterações no arquivo **index.html**, como modificar o texto dentro das tags **<h1>**, **<p>** ou adicionando novos elementos HTML. Observe como o Live Server atualiza automaticamente a página no navegador, demonstrando como as alterações na árvore DOM são refletidas na renderização da página.

Eventos HTML e a Importância da Documentação

Eventos HTML são como convites para interagir em uma festa na web! Eles permitem que sua página da web responda quando alguém faz algo, como clicar em um botão ou mover o mouse sobre uma imagem. É como dar vida à sua página! Você já deve ter visto aqueles botões que mudam de cor quando você passa o mouse sobre eles. Isso é um evento! E você pode fazer isso acontecer no seu site também.

Vamos entender isso melhor: quando você cria uma página web, pode adicionar esses "convites" aos elementos HTML. Por exemplo, se você quiser que algo aconteça quando alguém clicar em um botão, pode adicionar um evento **onclick** a esse botão. Dentro desse evento, você pode colocar o que quiser que aconteça, como mostrar uma mensagem ou fazer algo acontecer na página.

```
<button onclick="console.log('O botão foi clicado!')>Clique aqui</button>
```

Neste exemplo, quando o botão é clicado, a mensagem "O botão foi clicado!" será registrada no console do navegador. Ah, e uma ferramenta muito legal que você pode usar dentro desses eventos é o **console.log**. Ele é como um gravador que escreve mensagens que só você pode ver, no "porão" da sua página, onde ficam guardadas as informações do site. É muito útil para entender o que está acontecendo enquanto você desenvolve o seu site.

Eventos HTML são uma parte essencial da interatividade em páginas da web. Eles permitem que você responda a ações do usuário, como clicar em um botão, passar o mouse sobre uma imagem ou pressionar uma tecla no teclado. Esses eventos podem desencadear scripts ou executar funções específicas, tornando a experiência do usuário mais dinâmica e envolvente.

Existem vários tipos de eventos HTML, cada um correspondendo a uma ação específica do usuário. Alguns dos eventos mais comuns incluem:

- **onClick**: Este evento é acionado quando um elemento é clicado pelo usuário. É frequentemente usado em botões e links para executar uma ação quando o usuário os seleciona.
- **onMouseOver/onMouseOut**: Esses eventos são acionados quando o cursor do mouse entra ou sai de um elemento, respectivamente. Eles são úteis para criar efeitos de hover, como destacar uma imagem quando o mouse está sobre ela.
- **onSubmit**: Este evento é acionado quando um formulário é enviado pelo usuário, geralmente pressionando o botão "Enviar". É comumente usado para validar os dados do formulário antes de serem enviados para o servidor.
- **onKeyPress/onKeyDown/onKeyUp**: Esses eventos são acionados quando o usuário pressiona, mantém pressionada ou solta uma tecla no teclado, respectivamente. Eles são úteis para capturar entradas de texto em campos de formulário ou para criar atalhos de teclado em aplicativos da web.

Para utilizar eventos HTML em uma página da web, você pode adicionar atributos de evento aos elementos HTML relevantes.

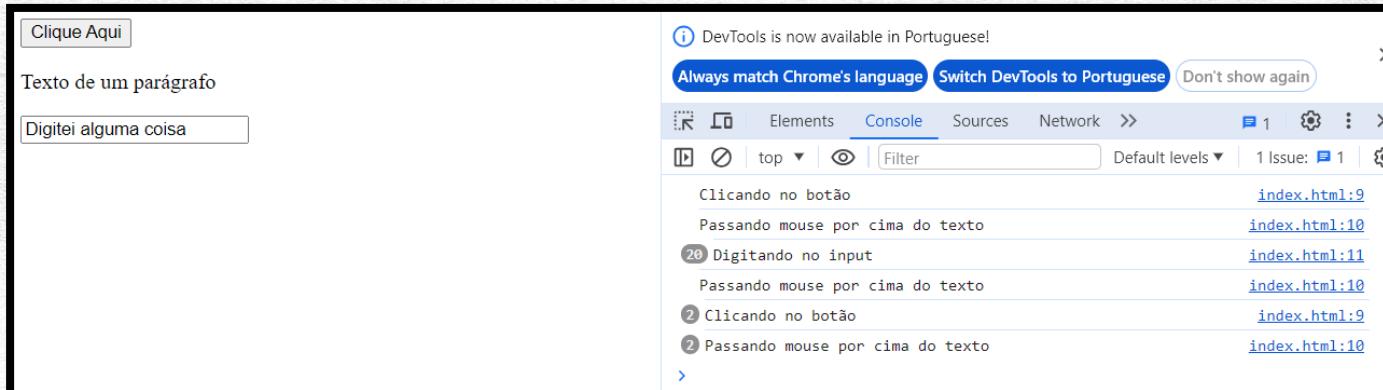
A documentação é como um mapa que mostra todos os caminhos que você pode seguir para criar coisas incríveis na web. Se você quer saber como um evento funciona, ou precisa de exemplos de código, ou mesmo se está procurando inspiração, a documentação é o lugar certo para ir.

Ler a documentação é como ter um guia especializado ao seu lado, garantindo que você saiba como usar cada ferramenta e que você esteja atualizado sobre as melhores práticas. Então, da próxima vez que estiver construindo algo na web, não se esqueça de consultar a documentação. ([HTML Event Attributes \(w3schools.com\)](https://www.w3schools.com/html/html_event_attributes.asp))

Agora vamos analisar o código da nossa aula:

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">
6      <title>Document</title>
7  </head>
8  <body>
9      <button onclick="console.log('Clicando no botão')>Clique Aqui</button>
10     <p onmouseover="console.log('Passando mouse por cima do texto')>Texto de um parágrafo</p>
11     <input type="text" placeholder="Digite Aqui" onkeypress="console.log('Digitando no input')">
12 </body>
13 </html>
```

Existem três interações diferentes nesta página web simples. Primeiro, ao clicar em um botão com o texto "Clique Aqui", a mensagem "Clicando no botão" é exibida no console. Segundo, mover o mouse sobre um parágrafo com o texto "Texto de um parágrafo" resulta na mensagem "Passando mouse por cima do texto" sendo exibida no console. Por último, ao digitar algo em um campo de texto com o marcador de posição "Digite Aqui", a mensagem "Digitando no input" é registrada e exibida no console.



- **Evento de Clique em Botão**

O evento de clique em um botão ocorre quando um usuário clica em um botão específico na página da web. Isso é útil para executar ações quando o usuário interage com o botão.

- **Evento de Passagem do Mouse**

O evento de passagem do mouse ocorre quando o cursor do mouse é movido sobre um elemento específico na página da web. Isso é útil para fornecer feedback visual ou acionar ações quando o usuário passa o mouse sobre um elemento.

- **Evento de Digitação em Campo de Texto**

O evento de digitação em um campo de texto ocorre quando o usuário digita algo em um campo de texto na página da web. Isso é útil para validar entradas do usuário ou executar ações com base no que está sendo digitado.

Agora vamos criar um arquivo script.js para criar funções que forneçam instruções aos nossos eventos citados acima, e vamos importar dentro do arquivo index.html o arquivo criado:

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">
6      <title>Document</title>
7  </head>
8  <body>
9      <button onclick="onClick()">Clique Aqui</button>
10     <p onmouseover="onMouseOver()">Texto de um parágrafo</p>
11     <input type="text" placeholder="Digite Aqui" onkeypress="onKeyPress()">
12     <script src="script.js"></script>
13 </body>
14 </html>
```

```
exemplo > JS script.js > ...
1  function onClick() {
2      console.log("Cliquei no botão");
3  }
4
5  function onMouseOver() {
6      console.log("Passando por cima do parágrafo");
7  }
8
9  function onKeyPress() {
10     console.log("Digitando no elemento input")
11 }
```



Separar as funcionalidades em um arquivo JavaScript e utilizar funções nos eventos HTML oferece várias vantagens:

Separação de preocupações (Separation of Concerns): Isso ajuda a manter seu código organizado e mais fácil de entender. Ao separar as funcionalidades em arquivos JavaScript dedicados, você mantém o HTML focado na estrutura e apresentação da página, enquanto o JavaScript é responsável pela lógica e interatividade.

Reutilização de código: Ao definir funções em um arquivo JavaScript, você pode reutilizá-las em vários elementos HTML. Por exemplo, se você tiver uma função **onMouseOver()** que faz algo sempre que o mouse passa sobre um elemento, você pode aplicá-la a vários elementos sem repetir o código.

Manutenção simplificada: Quando você precisa fazer alterações na lógica do seu código, é mais fácil e eficiente fazê-lo em um arquivo JavaScript separado. Isso evita a necessidade de procurar e modificar vários elementos HTML que possam ter os eventos embutidos.

Facilita a depuração (Debugging): Ao separar as funcionalidades em arquivos JavaScript, você pode usar ferramentas de depuração específicas para JavaScript para encontrar e corrigir erros mais facilmente. Isso é especialmente útil em projetos maiores, onde o código pode se tornar mais complexo.

Melhora a legibilidade do HTML: Manter os eventos embutidos diretamente no HTML pode tornar o código HTML confuso e difícil de ler, especialmente quando há muitos eventos. Separar as funções em um arquivo JavaScript torna o HTML mais limpo e fácil de entender.

No exemplo, ao mover as funções **onClick()**, **onMouseOver()**, e **onKeyPress()** para um arquivo JavaScript separado (**script.js**), estamos seguindo boas práticas de desenvolvimento, o que torna o código mais organizado, reutilizável e fácil de manter.

Quando falamos sobre manipulação de elementos HTML usando JavaScript, é essencial entender os diferentes tipos de métodos disponíveis para selecionar elementos na árvore do DOM (Document Object Model). A árvore do DOM é essencialmente uma representação hierárquica dos elementos HTML em uma página web, onde cada elemento é um nó na árvore.

Vamos começar discutindo os métodos para selecionar vários elementos:

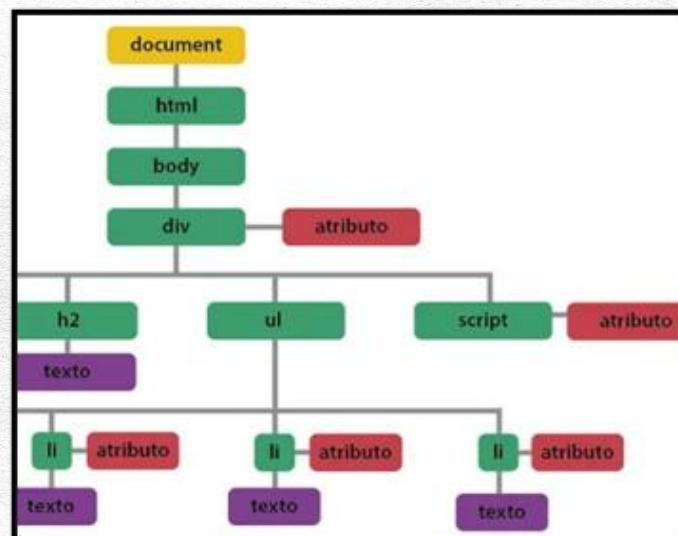
- **getElementsByTagName("tag")**: Este método seleciona elementos pelo nome da tag. Por exemplo, se você quiser selecionar todos os elementos **<p>** em uma página, você pode usar **document.getElementsByTagName("p")**.
- **getElementsByClassName("classe")**: Este método seleciona elementos por classe. Se você tem elementos com a mesma classe, pode usar **document.getElementsByClassName("classe")** para selecioná-los.
- **querySelectorAll(".classe")**: Aqui, você pode selecionar todos os elementos que correspondem a um seletor CSS específico. Por exemplo, **document.querySelectorAll(".classe")** selecionará todos os elementos com a classe especificada.
- **querySelectorAll("*")**: Este método seleciona todos os elementos no documento. Simplesmente usando **document.querySelectorAll("*")**, você pode obter todos os elementos presentes na página.
- **getElementsByName("atributoName")**: Este método seleciona elementos pelo atributo **name**. Se você tiver elementos com um atributo **name** específico, pode usar **document.getElementsByName("atributoName")**.

Agora, vamos falar sobre métodos para selecionar elementos únicos:

getElementById("id"): Este é um dos métodos mais comuns e eficientes para selecionar um elemento único pelo seu ID. Por exemplo, se você tem um elemento com o ID **myElement**, você pode selecioná-lo usando **document.getElementById("myElement")**.

querySelector("#id"): Similar ao método anterior, este método seleciona um elemento usando um seletor CSS. Para selecionar um elemento pelo seu ID, você pode usar **document.querySelector("#id")**.

Imagine a árvore do DOM como uma estrutura de árvore onde cada nó é um elemento HTML. Os métodos de seleção de elementos mencionados anteriormente são como ferramentas que permitem percorrer essa árvore e selecionar os nós desejados com base em critérios específicos, como tags, classes, IDs ou seletores CSS. Esses métodos ajudam os desenvolvedores a interagir dinamicamente com os elementos de uma página web usando JavaScript.



Agora, vamos aprender sobre algumas tags de HTML que serão úteis no nosso código. Essas tags nos ajudarão a aplicar os métodos de seleção de elementos do DOM.

- **<header>**: Esta tag é usada para definir o cabeçalho de uma página ou de uma seção dentro dela. O conteúdo dentro do elemento `<header>` geralmente inclui elementos de navegação, logotipos, links de retorno à página inicial, etc. Em resumo, o `<header>` é onde você coloca informações importantes que aparecem no topo da página ou no topo de uma seção.
- **<h1>**: Esta é uma tag usada para definir um cabeçalho de nível 1 em HTML. Ela representa o título principal da página ou de uma seção específica. O `<h1>` é frequentemente usado uma vez por página e é o cabeçalho mais importante e geralmente o maior em tamanho. Os motores de busca tendem a dar um peso considerável ao conteúdo dentro de `<h1>` porque geralmente é o título principal do conteúdo.
- **<main>**: Esta tag é usada para envolver o conteúdo principal de uma página. Ela identifica o conteúdo principal que é exclusivo para aquela página em particular. Isso significa que dentro da tag `<main>`, você encontrará o conteúdo principal da página, como artigos, posts de blog, informações principais, etc. É importante notar que deve haver apenas um elemento `<main>` em uma página HTML.
- **<section>**: Esta tag é usada para agrupar conteúdo relacionado em uma página HTML. Ela permite dividir o conteúdo em seções distintas e identificar claramente diferentes partes da página. Dentro da tag `<section>`, você colocará conteúdo que está semanticamente relacionado, como uma seção de produtos, uma seção de depoimentos de clientes, ou uma seção de recursos. Cada `<section>` pode ter seu próprio cabeçalho, rodapé e hierarquia de títulos, tornando-a independente e fácil de entender. É importante usar a tag `<section>` de forma significativa para melhorar a estrutura e a acessibilidade da página.

- **<h2>**: A tag **<h2>** é usada em HTML para definir um cabeçalho de segundo nível. Ela é usada para títulos secundários em uma página, geralmente menores e subordinados aos títulos principais definidos com **<h1>**. Por exemplo, se você tem um título principal usando **<h1>** como "Serviços", os subitens ou categorias dentro dos serviços podem ser marcados como **<h2>**. A hierarquia de títulos ajuda a estruturar o conteúdo e é importante para a acessibilidade e SEO.
- **<p>**: A tag **<p>** é usada para definir um parágrafo de texto em HTML. Ela é usada para agrupar blocos de texto em parágrafos distintos. Qualquer texto dentro da tag **<p>** é considerado um novo parágrafo. Por exemplo, você pode usar **<p>** para separar diferentes ideias ou seções de texto em um documento HTML.
- **<div>**: A tag **<div>** é uma das mais versáteis em HTML. Ela é usada como um contêiner genérico para agrupar e estilizar elementos em blocos distintos. A tag **<div>** não possui significado semântico próprio, mas é frequentemente usada junto com classes ou IDs para aplicar estilos CSS ou para fins de organização do código HTML. Por exemplo, você pode usar **<div>** para agrupar elementos relacionados, como uma seção de conteúdo em uma página web.
- ****: A tag **** é usada para criar listas não ordenadas em HTML. Dentro de uma lista não ordenada, você pode incluir elementos de lista marcados com **** (item de lista). Os itens de lista dentro de uma **** geralmente são exibidos com marcadores, como pontos. Este tipo de lista é comumente usado para itens que não precisam ser ordenados em uma sequência específica, como uma lista de recursos, pontos de destaque, etc.

<input>: Esta tag é usada em HTML para criar uma área interativa onde os usuários podem inserir dados. Aqui estão algumas propriedades específicas dessa tag:

- **Type**: A propriedade "type" define o tipo de entrada que o elemento **<input>** representa. No caso de **<input type="text">**, estamos criando um campo de entrada de texto simples.
- **Placeholder**: A propriedade "placeholder" permite que você forneça um texto de exemplo dentro do campo de entrada. Esse texto desaparece quando o usuário começa a digitar, fornecendo uma dica sobre o que é esperado no campo.
- **Name**: A propriedade "name" é usada para identificar o campo de entrada quando o formulário é submetido. Cada campo em um formulário deve ter um nome único para que os dados possam ser identificados corretamente pelo servidor.

ID (Identificador):

- Um ID é um atributo único atribuído a um elemento HTML para identificação exclusiva.
- Cada elemento em uma página HTML pode ter no máximo um ID único.
- Em HTML, o ID é definido no atributo **id** de um elemento. Por exemplo: **<div id="header">...</div>**.

Classe:

- Uma classe é um atributo que pode ser atribuído a um ou mais elementos HTML para agrupá-los ou estilizá-los de maneira semelhante.
- Um elemento HTML pode ter várias classes.
- Em HTML, a classe é definida no atributo **class** de um elemento. Por exemplo: **<div class="destaque">...</div>**.

Após compreendermos os conceitos das tags, vamos agora aplicá-las dentro de um novo arquivo chamado index.html. Vamos utilizar o Live Server para visualizar a disposição dos elementos.

```
8 <body>
9   <header>
10    |   <h1>Seleção de Elementos</h1>
11   |</header>
12
13   <main>
14     <section id="content">
15       <h2>Tipo de Elemento</h2>
16       <h2>Outro título</h2>
17       <p class="texto-lista">Parágrafo 1</p>
18       <p class="texto-lista">Parágrafo 2</p>
19       <div>
20         <ul>
21           <li>Item 1</li>
22           <li class="texto-lista">Item 2</li>
23           <li class="texto-lista">Item 3</li>
24         </ul>
25       </div>
26       <div>
27         <input type="text" placeholder="Nome de usuário" name="username"/>
28         <input type="text" placeholder="Endereço de email" name="email" />
29       </div>
30     </section>
31   </main>
32 </body>
```

Seleção de Elementos

Tipo de Elemento

Outro título

Parágrafo 1

Parágrafo 2

- Item 1
- Item 2
- Item 3

Nome de usuário

Endereço de email



Vamos criar nosso arquivo style.css que será responsável por armazenar a estilação que vamos criar para o nosso programa. O atributo **link** com **rel="stylesheet"** é usado em HTML para vincular um documento CSS (folha de estilos em cascata) ao documento HTML atual. Aqui está uma explicação detalhada de como funciona:

Estrutura básica do elemento link:

```
<link rel="stylesheet" type="text/css" href="caminho_para_seu_arquivo_css.css">
```

- **rel="stylesheet":**

Este atributo define o relacionamento entre o documento HTML atual e o documento CSS vinculado. No caso do **rel="stylesheet"**, ele indica que o documento CSS é uma folha de estilos para o documento HTML.

- **type="text/css":**

Este atributo especifica o tipo MIME do documento vinculado. Em documentos HTML, o tipo padrão de uma folha de estilos CSS é **text/css**. No entanto, em HTML5, esse atributo é opcional.

- **href="caminho_para_seu_arquivo_css.css":**

Este atributo especifica o caminho para o arquivo CSS que você deseja vincular ao documento HTML. O valor de **href** deve ser o caminho relativo ou absoluto para o arquivo CSS.

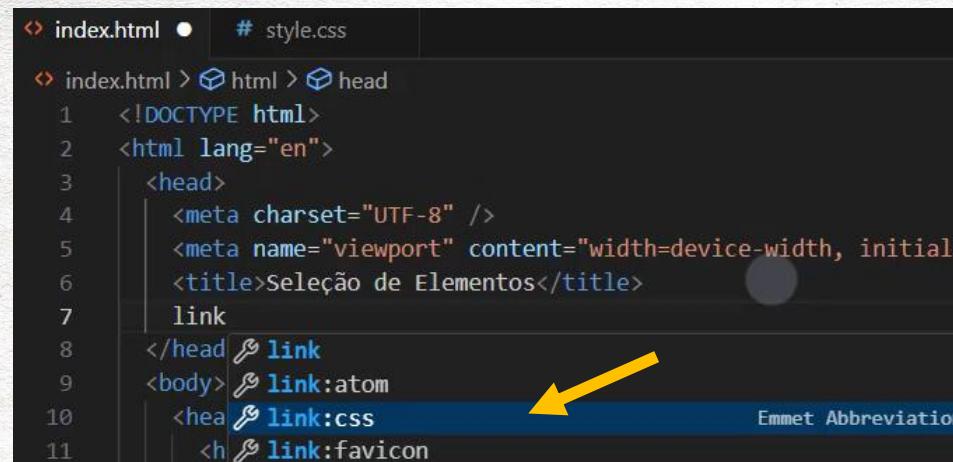
Como funciona:

Quando o navegador encontra o elemento **link** com **rel="stylesheet"** no documento HTML, ele faz uma solicitação HTTP para o arquivo CSS especificado no atributo **href**. Uma vez que o navegador carrega o arquivo CSS, ele aplica as regras de estilo contidas nele ao conteúdo HTML, formatando a aparência do documento de acordo com essas regras.

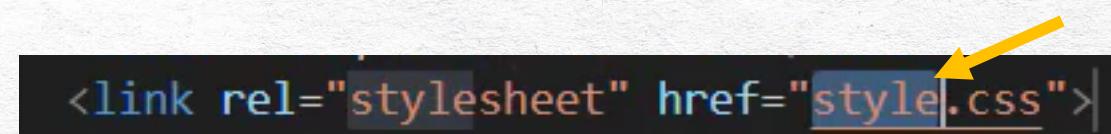
Benefícios:

- **Separação de preocupações:** Mantém o conteúdo HTML separado da apresentação visual, o que facilita a manutenção e a atualização do estilo do site.
- **Reutilização de estilos:** Permite que múltiplas páginas HTML compartilhem a mesma folha de estilos CSS, promovendo a consistência visual em todo o site.
- **Facilidade de atualização:** Alterações no estilo do site podem ser feitas apenas no arquivo CSS, sem a necessidade de modificar o conteúdo HTML.

É importante garantir que o caminho especificado no atributo **href** esteja correto para que o navegador possa encontrar e carregar o arquivo CSS corretamente.



```
index.html # style.css
index.html > html > head
1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="UTF-8" />
5      <meta name="viewport" content="width=device-width, initial-
6      <title>Seleção de Elementos</title>
7      link
8    </head> ↵ link
9    <body> ↵ link:atom
10   <head> ↵ link:css
11     <h1> ↵ link:favicon
```

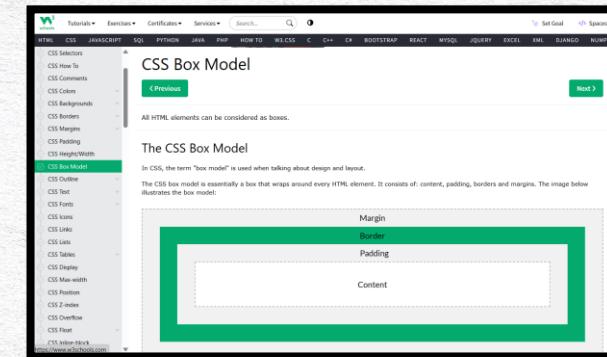


Módulo 9 – Selecionando elementos no DOM – parte 3 (3 / 8)

O W3Schools é um recurso incrível para quem está aprendendo desenvolvimento web, especialmente para iniciantes. Você encontrará tutoriais e documentação sobre CSS (Cascading Style Sheets), que é uma linguagem de estilo usada para estilizar o conteúdo HTML em páginas da web.

[CSS Tutorial \(w3schools.com\)](#)

Aqui estão algumas das coisas que você pode aprender no W3Schools sobre CSS:



- **Seletores e Propriedades:** Você aprenderá sobre os diferentes tipos de seletores CSS, que permitem direcionar elementos HTML específicos, e sobre as diversas propriedades que podem ser aplicadas a esses elementos para controlar sua aparência.
- **Box Model:** O box model é uma parte crucial do CSS. Ele descreve como os elementos HTML são renderizados como caixas retangulares, e como você pode controlar as dimensões e o espaçamento dessas caixas usando propriedades como width, height, padding e margin.
- **Posicionamento:** Você aprenderá sobre as diferentes técnicas de posicionamento em CSS, como static, relative, absolute e fixed, que permitem posicionar elementos de forma precisa na página.
- **Layouts Responsivos:** O W3Schools oferece informações sobre como criar layouts responsivos que se adaptam a diferentes tamanhos de tela, o que é essencial para garantir uma boa experiência do usuário em dispositivos móveis e desktops.
- **Animações e Transições:** Você também pode aprender sobre como adicionar animações e transições aos elementos HTML usando CSS, o que pode tornar sua página mais dinâmica e envolvente para os visitantes.

Esses são apenas alguns dos tópicos que você encontrará no W3Schools sobre CSS. É um recurso valioso para iniciantes, pois fornece explicações claras e exemplos práticos que ajudam a solidificar o entendimento dos conceitos.

Como podemos construir a nossa folha de estilo:

- **Estilos de Elemento:**

- Ao aplicar estilos usando elementos, você está afetando todos os elementos desse tipo em sua página HTML.
- Por exemplo, se você definir um estilo para o elemento **<h2>**, todos os títulos h2 em sua página terão esse estilo aplicado.
- É como pintar todos os carros vermelhos em uma garagem cheia de carros.

- **Estilos de Classe:**

- Com classes, você pode agrupar elementos e aplicar estilos a esse grupo específico.
- Pode usar a mesma classe em múltiplos elementos, e eles compartilharão os estilos definidos para essa classe.
- Por exemplo, se você criar uma classe chamada "destaque", todos os elementos com essa classe terão os estilos de destaque que você definiu.
- É como dar uma jaqueta vermelha para todos que estão no clube de corrida.

- **Estilos de ID:**

- Os IDs são usados para identificar elementos específicos em sua página.
- Cada ID deve ser único em sua página HTML.
- Com IDs, você pode aplicar estilos a um único elemento.
- Por exemplo, se você tem um elemento especial que deseja estilizar de forma única, você pode usar um ID para isso.
- É como dar uma placa de identificação única para uma pessoa em uma festa, para que todos saibam quem é essa pessoa.

header (Cabeçalho):

- **background-color: #333;**: Define a cor de fundo do elemento **<header>** como um tom de cinza escuro (#333).
- **color: white;**: Define a cor do texto dentro do elemento **<header>** como branco.
- **text-align: center;**: Alinha o texto dentro do elemento **<header>** ao centro.
- **padding: 20px 0;**: Define o preenchimento (espaço entre o conteúdo e a borda) do elemento **<header>**. Neste caso, 20 pixels de preenchimento nas partes superior e inferior e nenhum preenchimento nas laterais.

h1 (Título de Nível 1):

- **margin: 0;**: Remove qualquer margem ao redor do elemento **<h1>**, ou seja, o título não terá espaço extra ao seu redor.
- **font-size: 28px;**: Define o tamanho da fonte do texto dentro do elemento **<h1>** como 28 pixels.

main (Seção Principal):

- **width: 80%;**: Define a largura do elemento **<main>** como 80% da largura do seu contêiner pai.
- **margin: 20px auto;**: Define as margens do elemento **<main>**. Neste caso, 20 pixels de margem em cima e embaixo, e "auto" nas laterais para centralizar o elemento horizontalmente na página.

h2 (Título de Nível 2):

- **font-size: 24px;**: Define o tamanho da fonte do texto dentro do elemento **<h2>** como 24 pixels.
- **margin-bottom: 10px;**: Define a margem inferior do elemento **<h2>** como 10 pixels, deixando um espaço extra abaixo do título.
- **color: #333;**: Define a cor do texto dentro do elemento **<h2>** como um tom de cinza escuro (#333).

```
1 header {  
2     background-color: □ #333;  
3     color: □ white;  
4     text-align: center;  
5     padding: 20px 0;  
6 }  
7  
8 h1 {  
9     margin: 0;  
10    font-size: 28px;  
11 }  
12  
13 main {  
14     width: 80%;  
15     margin: 20px auto;  
16 }  
17  
18 h2 {  
19     font-size: 24px;  
20     margin-bottom: 10px;  
21     color: □ #333;  
22 }
```



.texto-lista (Texto da Lista):

- **font-style: italic;**: Define o estilo da fonte do texto dentro dos elementos com a classe **texto-lista** como itálico.
- **margin-bottom: 15px;**: Define a margem inferior dos elementos com a classe **texto-lista** como 15 pixels, deixando um espaço extra abaixo do texto.

#destacado (Elemento Destacado):

- **background-color: yellow;**: Define a cor de fundo do elemento com o ID **destacado** como amarelo.
- **padding: 10px;**: Define o preenchimento (espaço entre o conteúdo e a borda) do elemento com o ID **destacado** como 10 pixels em todas as direções.
- **margin-bottom: 15px;**: Define a margem inferior do elemento com o ID **destacado** como 15 pixels, deixando um espaço extra abaixo deste elemento.

```
24 .texto-lista {  
25   font-style: italic;  
26   margin-bottom: 15px;  
27 }  
28  
29 #destacado {  
30   background-color: yellow;  
31   padding: 10px;  
32   margin-bottom: 15px;  
33 }
```

ul (Lista não Ordenada):

- **list-style: none;**: Remove os marcadores de lista padrão (pontos ou números) dos elementos de lista não ordenada (****).
- **padding: 0;**: Remove o preenchimento padrão das listas não ordenadas, removendo qualquer espaço entre a borda do elemento **** e seus elementos filhos (****).

div (Divisão de Conteúdo):

- **display: flex;**: Define a divisão (**<div>**) como um contêiner flexível, permitindo um layout flexível de seus elementos filhos. Isso geralmente é usado para organização e posicionamento de elementos dentro da divisão.

input (Campo de Entrada):

- **margin-left: 10px;**: Define a margem à esquerda dos campos de entrada (**<input>**), criando um espaço extra entre o campo e os elementos à sua esquerda.
- **margin-bottom: 10px;**: Define a margem inferior dos campos de entrada como 10 pixels, deixando um espaço extra abaixo deles.
- **padding: 5px;**: Define o preenchimento (espaço interno) dos campos de entrada como 5 pixels em todas as direções, criando um espaço entre o texto do campo e sua borda.

```
35  ul {  
36    list-style: none;  
37    padding: 0;  
38  }  
39  
40  div {  
41    display: flex;  
42  }  
43  
44  input {  
45    margin-left: 10px;  
46    margin-bottom: 10px;  
47    padding: 5px;  
48  }
```

Após aplicarmos essas propriedades nos elementos, classes e id teremos a folha de estilo aplicada em nosso programa:

The screenshot displays a web page with the following structure and content:

- Section Header:** **Seleção de Elementos**
- Section:** **Tipo de Elemento (h2)**
- Section:** **Outro título (h2)**
- Text:** *Parágrafo 1*
- Text:** *Parágrafo 2*
- List:** Item 1
Item 2
Item 3
- Form Fields:**

A large yellow rectangular area highlights the list of items (Item 1, Item 2, Item 3) under the "Outro título (h2)" section.

Quando você quer importar um arquivo JavaScript em um arquivo HTML, você pode usar a tag **<script>**. O atributo **src** é usado para especificar o URL do arquivo JavaScript que você deseja importar.

```
32      </main>
33      <script src="script.js"></script>
34    </body>
```

- **<script src="script.js"></script>**: Esta linha importa o arquivo JavaScript chamado **script.js**. O navegador vai procurar esse arquivo no mesmo diretório em que o arquivo HTML está localizado. Se o arquivo **script.js** estiver em um diretório diferente, você precisa especificar o caminho correto no atributo **src**.

Quando o navegador encontra essa tag **<script>**, ele faz uma solicitação para o servidor para baixar o arquivo **script.js** e, em seguida, executa o código JavaScript contido nele. Isso permite que você use as funções e variáveis definidas em **script.js** no seu arquivo HTML.

Ao utilizar o método **document**. em JavaScript para selecionar elementos HTML, estamos basicamente acessando o Document Object Model (DOM) da página. O DOM é uma interface de programação para documentos HTML e XML, onde cada elemento na página é representado como um nó em uma estrutura de árvore.

Quando utilizamos **document**, estamos referenciando o objeto principal que representa a estrutura do documento HTML atual. Com isso, podemos acessar e manipular qualquer elemento presente na página, como parágrafos, imagens, botões, etc.

Quando selecionamos elementos HTML usando o método **document**. em JavaScript, estamos essencialmente criando referências a esses elementos. Essas referências são essenciais para que possamos manipular esses elementos de forma eficiente e dinâmica. Ao armazenar essas referências em variáveis, estamos criando uma forma conveniente de acessar e trabalhar com esses elementos posteriormente em nosso código. Por exemplo, podemos armazenar a referência de um elemento em uma variável e, em seguida, usar essa variável para acessar suas propriedades, modificar seu conteúdo, aplicar estilos e até mesmo adicionar ou remover o elemento do DOM.

Enquanto o terminal do VS Code pode ser útil para depuração em certos contextos, como em aplicativos Node.js, para o desenvolvimento web é geralmente mais eficaz utilizar as ferramentas de desenvolvedor do navegador para observar as saídas do **console.log()** e depurar o código JavaScript. Então vamos utilizar o DevTools para inspecionar o nosso código.

O método **getElementsByName()** é uma função em JavaScript que é usada para selecionar elementos HTML em uma página da web com base em suas tags. Quando você usa **document.getElementsByName("h2")**, está instruindo o navegador a encontrar todos os elementos **<h2>** na página atual.

```
1 // getElementsByName: Seleciona elementos pelo nome da tag.  
2 const titulo = document.getElementsByName("h2");  
3 console.log("Selecionando pela tag h2: ", titulo);
```

O método **getElementsByClassName()** é uma ferramenta poderosa em JavaScript, frequentemente usada para selecionar elementos específicos em uma página da web com base em suas classes CSS. Quando você utiliza **document.getElementsByClassName("texto-lista")**, está instruindo o navegador a encontrar todos os elementos que possuem a classe CSS "**texto-lista**".

```
5 // getElementsByClassName: Seleciona elementos por classe.  
6 const paragrafosItens = document.getElementsByClassName("texto-lista");  
7 console.log("Selecionando por nome de classe: ", paragrafosItens);
```

O método **getElementById()** é uma função fundamental em JavaScript usada para selecionar um único elemento HTML com base em seu atributo **id**. Quando você utiliza **document.getElementById("destacado")**, está instruindo o navegador a encontrar o elemento que possui o atributo **id** com o valor "**destacado**".

```
9 // getElementById: Seleciona um elemento pelo ID.  
10 const destacado = document.getElementById("destacado");  
11 console.log("Selecionando por ID: ", destacado);  
12
```



Módulo 9 – Selecionando elementos no DOM – parte 4 (4 / 4)

O método **querySelector()** é uma função em JavaScript que permite selecionar um elemento HTML usando seletores CSS. Quando você usa **document.querySelector("#destacado")**, está instruindo o navegador a encontrar o primeiro elemento que corresponde ao seletor CSS "**#destacado**", onde **#destacado** representa um seletor de ID.

```
13 // querySelector: Seleciona um elemento usando um seletor CSS.
14 const destacadoSelector = document.querySelector("#destacado");
15 console.log("Selecionando ID por seletor CSS: ", destacadoSelector);
```

O método **querySelectorAll()** é semelhante ao **querySelector()**, porém em vez de retornar apenas o primeiro elemento que corresponde ao seletor CSS especificado, ele retorna todos os elementos que correspondem ao seletor. Quando você usa **document.querySelectorAll(".texto-lista")**, está instruindo o navegador a encontrar todos os elementos que possuem a classe CSS "**texto-lista**".

```
17 // querySelectorAll: Seleciona todos os elementos que correspondem a um seletor CSS.
18 const todosPItens = document.querySelectorAll(".texto-lista");
19 console.log("Selecionando múltiplos elementos por seletor CSS: ", todosPItens);
```

O método **getElementsByName()** é usado para selecionar elementos HTML com base em seus atributos **name**. Ao usar **document.getElementsByName('username')**, você está instruindo o navegador a encontrar todos os elementos que têm o atributo **name** definido como '**username**'.

```
21 // getElementByName: Seleciona elementos pelo atributo name.
22 const inputName = document.getElementsByName('username');
23 console.log("Selecionando por atributo name (username): ", inputName),
```

O método **querySelectorAll()** com o seletor **"**"** é usado para selecionar todos os elementos HTML presentes na página. Quando você usa **document.querySelectorAll("")**, está instruindo o navegador a encontrar todos os elementos HTML, independente de suas tags, classes ou ids.

```
25 // querySelectorAll(): Seleciona todos os elementos no documento.
26 const todosElementos = document.querySelectorAll("");
27 console.log("Selecionando TODOS os elementos: ", todosElementos);
```



Nessa aula vamos entender os conceitos `HTMLCollection` e `NodeList`, para isso vamos iniciar com o seguinte código em um arquivo chamado "index.html".

```
index.html > html
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">
6      <title>Coleções</title>
7  </head>
8  <body>
9      <h1 style="font-size: large;">Acessando elementos do HTMLCollection e NodeList</h1>
10     <h2 style="font-size: small; color: blue;">HTMLCollection</h2>
11     <ul id="lista">
12         <li>Item 1</li>
13         <li>Item 2</li>
14         <li>Item 3</li>
15     </ul>
16
17     <h2 style="font-size: small; color: red;">NodeList</h2>
18     <div id="conteudo">
19         <p>Parágrafo 1</p>
20         <p>Parágrafo 2</p>
21         <p>Parágrafo 3</p>
22     </div>
23 </body>
24 </html>
```

Acessando elementos do `HTMLCollection` e `NodeList`

`HTMLCollection`

- Item 1
- Item 2
- Item 3

`NodeList`

Parágrafo 1

Parágrafo 2

Parágrafo 3



Entendendo HTMLCollection e NodeList

Quando começamos a explorar o mundo da programação web, nos deparamos com diversos conceitos e estruturas que podem parecer confusos no início. Dois desses conceitos são HTMLCollection e NodeList, que são frequentemente mencionados ao trabalhar com JavaScript para manipulação do DOM (Modelo de Objeto de Documento).

O que são HTMLCollection e NodeList?

Tanto HTMLCollection quanto NodeList são objetos usados para armazenar conjuntos de elementos HTML. Eles são semelhantes, mas têm algumas diferenças importantes.

HTMLCollection:

- Um HTMLCollection é uma coleção (ou lista) de elementos HTML, como elementos `<div>`, `<p>`, `<a>`, entre outros, que são coletados de um documento HTML.
- Um HTMLCollection é uma coleção viva, o que significa que ele é atualizado automaticamente conforme o documento HTML é modificado.
- Você pode acessar os elementos de um HTMLCollection usando um índice numérico, como `myCollection[0]`, `myCollection[1]`, e assim por diante.

NodeList:

- Um NodeList também é uma coleção de elementos HTML, semelhante ao HTMLCollection.
- No entanto, um NodeList pode conter não apenas elementos HTML, mas também outros tipos de nós do DOM, como nós de texto e nós de comentário.
- Assim como o HTMLCollection, um NodeList é uma coleção viva e é atualizado conforme o documento é modificado.
- Você também pode acessar os elementos de um NodeList usando um índice numérico, como `my NodeList[0]`, `my NodeList[1]`, e assim por diante.



Quando usar HTMLCollection e NodeList?

A escolha entre HTMLCollection e NodeList depende da situação específica em que você está trabalhando. Em muitos casos, eles podem ser usados de forma intercambiável. No entanto, se você precisa acessar não apenas elementos HTML, mas também outros tipos de nós do DOM, como nós de texto, um NodeList é a melhor opção.

HTMLCollection:

- **Conceito:** Uma HTMLCollection é uma coleção (ou lista) de elementos HTML, que representa uma coleção ao vivo dos elementos que correspondem a um critério específico no documento HTML. Essa coleção é dinâmica, o que significa que se os elementos no documento forem alterados, a HTMLCollection será automaticamente atualizada para refletir essas mudanças.
- **Obtenção:** Geralmente, uma HTMLCollection é retornada por métodos como `getElementsByName`, `getElementsByClassName`, e `children`.
- **Indexação:** É indexada numericamente, permitindo acessar os elementos pelo seu índice numérico (como uma array).

NodeList:

- **Conceito:** Um NodeList é uma lista de nós (nodes) do DOM, que pode conter vários tipos diferentes de nós, como elementos HTML, textos e nós de comentário. Similar à HTMLCollection, é uma lista ao vivo, portanto, se houver alterações no DOM, o NodeList será atualizado automaticamente.
- **Obtenção:** Geralmente, um NodeList é retornado por métodos como `querySelectorAll` e várias propriedades do DOM, como `childNodes`.
- **Acesso aos elementos:** Pode-se acessar os elementos usando a numeração dos índices (como um array) ou percorrendo os nós com loops.

Diferenças entre `HTMLCollection` e `NodeList`:

Tipo de elementos: A `HTMLCollection` contém apenas elementos HTML, enquanto o `NodeList` pode conter diferentes tipos de nós do DOM (não apenas elementos HTML).

Métodos de obtenção: Geralmente, métodos como `getElementsByName` e `getElementsByClassName` retornam uma `HTMLCollection`, enquanto métodos como `querySelectorAll` retornam um `NodeList`.

Atualização dinâmica: Ambos são coleções dinâmicas, mas a origem da sua criação pode variar. A `HTMLCollection` pode ser mais específica ao tipo de elemento, enquanto o `NodeList` pode ser mais amplo, incluindo diferentes tipos de nós.

Acessibilidade: Ambos permitem o acesso aos elementos, mas os métodos para acessar e manipular os elementos podem ser ligeiramente diferentes, embora sejam semelhantes em muitos aspectos.

Em resumo, ambas as coleções são estruturas semelhantes que representam elementos do DOM, mas com diferenças sutis em sua composição e maneiras de serem obtidas. A escolha entre `HTMLCollection` e `NodeList` pode depender da necessidade específica de acesso aos elementos e do método utilizado para obter essas coleções.

Módulo 9 – HTML Collection versus NodeList (5 / 7)

Acessando e visualizando no DevTools:

```

26 <script>
27   const listaItens = document
28     .getElementById("lista")
29     .getElementsByTagName("li");
30
31   const paragrafos = document
32     .getElementById("conteudo")
33     .querySelectorAll("p");
34 </script>
```

Acessando elementos do HTMLCollection e NodeList

HTMLCollection

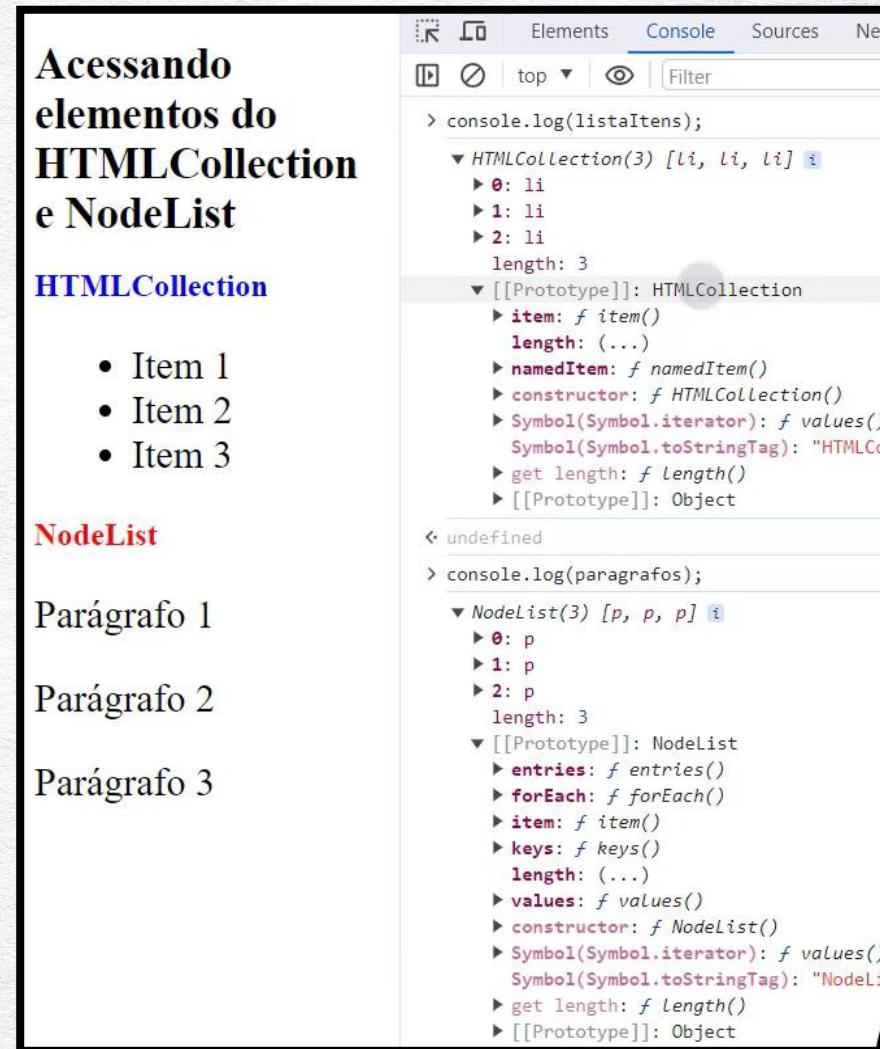
- Item 1
- Item 2
- Item 3

NodeList

Parágrafo 1

Parágrafo 2

Parágrafo 3



Acessando e visualizando no DevTools:

```
index.html > html > body > script
22   <p>Parágrafo 2</p>
23   <p>Parágrafo 3</p>
24 </div>
25
26 <script>
27   const listaItens = document
28     .getElementById("lista")
29     .getElementsByTagName("li");
30
31 //Acessar o segundo elemento da HTMLCollection
32 console.log("Segundo elemento: ", listaItens[1]);
33
34 const paragrafos = document
35   .getElementById("conteudo")
36   .querySelectorAll("p");
37
38 //Acessar os elementos do Nodelist usando loop - for
39 console.log("Acessando elementos do Nodelist lopp for: ");
40 for (let i = 0; i < paragrafos.length; i++) {
41   console.log(paragrafos[i]);
42 }
43
44 //Acessar o segundo elemento do Nodelist
45 console.log("Acessando o segundo elemento Nodelist: ", paragrafos[1]);
46 </script>
47 </body>
```

The screenshot shows a browser window with developer tools open. The left pane displays the code from index.html. The right pane shows the browser's output and the developer tools' console tab.

Browser Output:

- Segundo elemento: ... [index.html:32](#)
- Acessando elementos do Nodelist lopp for:
 - <p>Parágrafo 1</p> [index.html:41](#)
 - <p>Parágrafo 2</p> [index.html:41](#)
 - <p>Parágrafo 3</p> [index.html:41](#)
- Acessando o segundo elemento Nodelist:
 - <p>Parágrafo 2</p> [index.html:45](#)

Developer Tools Console:

Elements, Console, Sources, Network, etc. tabs are visible. The console log output matches the browser output.

Content of the page:

Acessando elementos do HTMLCollection e NodeList

HTMLCollection

- Item 1
- Item 2
- Item 3

NodeList

Parágrafo 1

Parágrafo 2

Parágrafo 3

Ambos permitem o acesso aos elementos por índices, assim como arrays, usando notação de colchetes `[]`, mas eles não possuem todos os métodos de arrays, como **forEach**, **map**, **filter**, etc. No entanto, você pode usar loops para iterar sobre eles e acessar cada elemento individualmente.

Isso ocorre porque NodeList e HTMLCollection não são realmente arrays; eles são objetos semelhantes a arrays que fornecem acesso a elementos pelo índice, mas não possuem todos os métodos e propriedades de arrays.

Nesta aula vamos aprender a criar elementos HTML utilizando lógica com Javascript. Vamos iniciar com um arquivo "index.html" com o seguinte conteúdo:

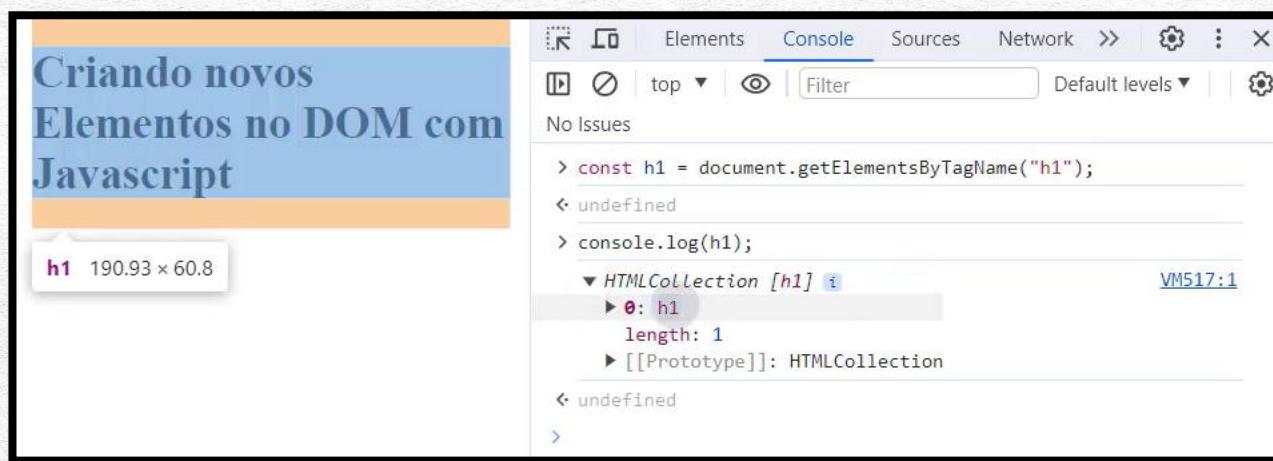
```
index.html > html
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">
6      <title>Criando novos elementos no DOM</title>
7  </head>
8  <body>
9      <h1 style="font-size: large;">Criando novos Elementos no DOM com Javascript</h1>
10     <button onclick="criarElemento()">Adicionar Elemento</button>
11     <ul id="lista-de-elementos"></ul>
12
13     <script src="script.js"></script>
14
15
16
17
```

1. Existe um botão na sua página HTML com o texto "Adicionar Elemento".
2. Quando alguém clica nesse botão, o evento **onclick** é ativado.
3. Esse evento está associado a uma função chamada **criarElemento()**.
4. O objetivo dessa função é criar elementos HTML dinamicamente.
5. Esses elementos criados serão adicionados dentro de uma lista **** com o id "lista-de-elementos".

Quando clicar no botão "Adicionar Elemento", a função **criarElemento()** é acionada e gera novos elementos HTML, que são então inseridos dentro da lista **** especificada.

Se você deseja acessar um elemento que já existe em uma página da web usando as ferramentas de desenvolvedor (DevTools), você pode fazer isso usando o Console. Aqui está um passo a passo básico:

- **Abra as Ferramentas de Desenvolvedor:** Você pode fazer isso pressionando F12 ou clicando com o botão direito do mouse na página e selecionando "Inspecionar" ou "Inspeção" no menu de contexto.
- **Acesse o Elemento no Console:** No console das Ferramentas de Desenvolvedor, você pode acessar o elemento usando JavaScript.



Agora vamos implementar a lógica dentro da nossa função criarElemento(). Dentro do arquivo script.js, vamos iniciar a nossa função e adicionar o armazenamento do nosso elemento ul:

```
1 ✓ function criarElemento() {  
2     const ul = document.getElementById("lista-de-elementos");  
3     //console.log(ul);]  
4 }
```

Document.createElement('elemento') é um método utilizado em JavaScript para criar um novo elemento HTML. Isso é útil quando você está construindo páginas da web dinamicamente e precisa adicionar elementos ao DOM (Modelo de Objeto de Documento).

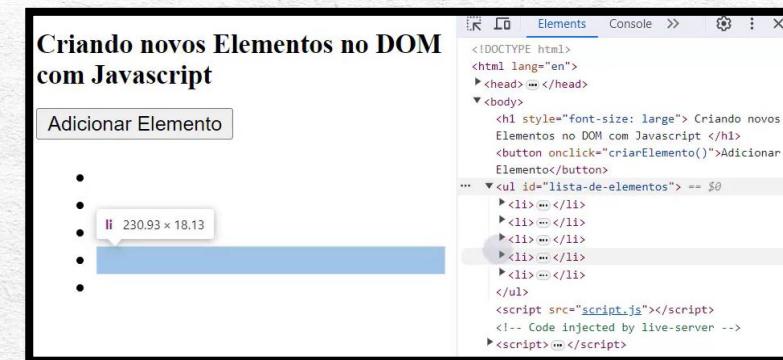
```
const novaLi = document.createElement("li");
```

O método **appendChild()** é usado em JavaScript para adicionar um nó (elemento) como filho de outro nó dentro do DOM (Modelo de Objeto de Documento). Ele basicamente insere um nó filho no final da lista de filhos do nó pai especificado.

Aqui está como funciona:

- **Seleção do nó pai:** Primeiro, você precisa selecionar o nó pai ao qual deseja adicionar o novo nó filho. Isso pode ser feito usando métodos como **getElementById**, **querySelector**, ou qualquer outra maneira de selecionar um elemento no DOM.
- **Criação do novo nó filho:** Em seguida, você cria o novo nó que deseja adicionar como filho ao nó pai. Isso pode ser feito usando o método **document.createElement()** para criar um novo elemento HTML, ou usando outras maneiras de criar nós, dependendo do tipo de nó que você deseja adicionar.
- **Adição do nó filho ao nó pai:** Por fim, você usa o método **appendChild()** no nó pai, passando o nó filho que você deseja adicionar como argumento. Isso colocará o nó filho como o último filho do nó pai, ou seja, o novo nó será adicionado ao final da lista de filhos do nó pai.

```
1 function criarElemento() {  
2     const ul = document.getElementById("lista-de-elementos");  
3     //console.log(ul);  
4  
5     const novaLi = document.createElement("li");  
6     //console.log("Adicionando novaLi");  
7     ul.appendChild(novaLi);  
8 }
```

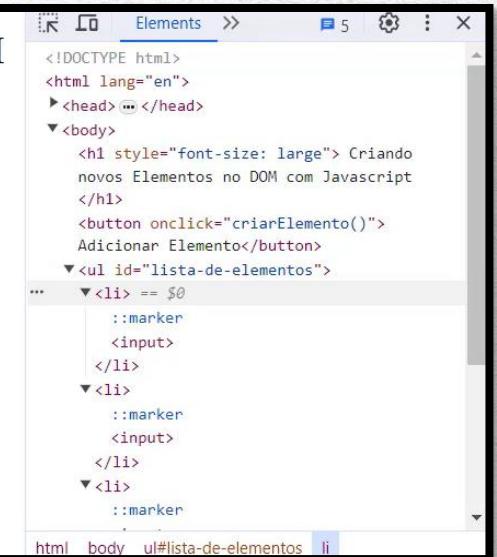
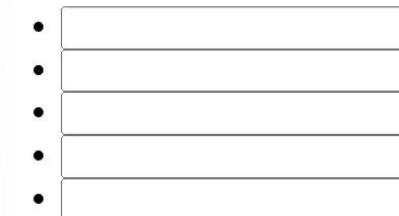


Também é possível implementar uma lógica para criar um novo elemento do tipo input dentro de um novo elemento li, que será criado ao clicar no botão.

```
1  function criarElemento() {  
2      const ul = document.getElementById("lista-de-elementos");  
3      //console.log(ul);]  
4  
5      const novaLi = document.createElement("li");  
6  
7      const novoInput = document.createElement("input");  
8      //console.log("Adicionando novos inputs e lis");  
9  
10     novaLi.appendChild(novoInput);  
11     ul.appendChild(novaLi);  
12 }
```

Criando novos Elementos no DOM com Javascript

Adicionar Elemento



A diferença entre **document.getElementByTagName("body")** e **document.body** está na forma como eles acessam o elemento **<body>** no HTML.

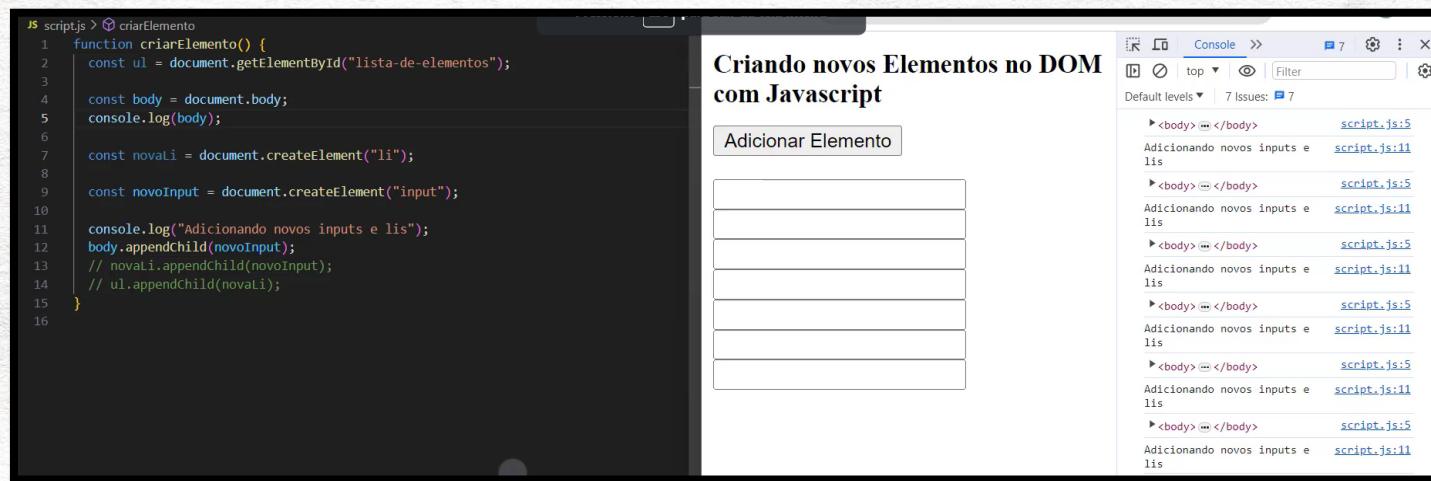
- **document.getElementByTagName("body")**: Esta função retorna uma coleção (array-like) de todos os elementos com o nome de tag especificado, no caso, "body". O retorno é uma NodeList, mesmo que só haja um elemento com esse nome.

document.body: Esta é uma propriedade que representa diretamente o elemento **<body>** no DOM. Ela é mais simples e direta, pois não requer a obtenção de uma coleção de elementos e a especificação de um índice. Portanto, para acessar o elemento **<body>**, seria necessário especificar o índice 0, pois a função retorna uma coleção.

Exemplo:

```
const body = document.getElementsByTagName("body")[0];
```

Em resumo, **document.body** é uma maneira mais conveniente e direta de acessar o elemento **<body>** no DOM, enquanto **document.getElementByTagName("body")** exige uma etapa adicional para acessar o mesmo elemento.



Continuando com o código da aula passada:

```
index.html > html
1   <!DOCTYPE html>
2   <html lang="en">
3   <head>
4     <meta charset="UTF-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <title>Criando novos elementos no DOM</title>
7   </head>
8   <body>
9     <h1 style="font-size: large;">Criando novos Elementos no DOM com Javascript</h1>
10    <button onclick="criarElemento()">Adicionar Elemento</button>
11    <ul id="lista-de-elementos"></ul>
12
13    <script src="script.js"></script>
14
15  </body>
16
17  </html>
```

Criando novos Elementos no DOM com Javascript

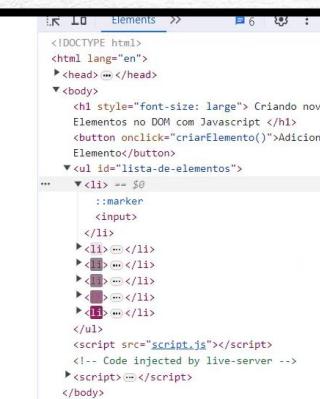
Adicionar Elemento

-
-
-
-
-
-

```
function criarElemento() {
  const ul = document.getElementById("lista-de-elementos");

  const novaLi = document.createElement("li");
  const novoInput = document.createElement("input");

  novaLi.appendChild(novoInput);
  ul.appendChild(novaLi);
}
```



Esses são três conceitos importantes em programação web relacionados à manipulação de conteúdo dentro de elementos HTML.

- **innerText**: O **innerText** é uma propriedade que permite acessar e modificar o conteúdo textual visível de um elemento HTML. Ele retorna o texto que é exibido na página, sem incluir qualquer marcação HTML que possa estar dentro desse elemento. Por exemplo, se você tem um parágrafo **<p>** com texto dentro dele, o **innerText** retornará apenas o texto desse parágrafo, sem quaisquer tags HTML.
- **textContent**: O **textContent** é semelhante ao **innerText**, mas com uma diferença crucial: ele retorna todo o texto dentro de um elemento, incluindo qualquer texto presente em elementos filhos, mas excluindo quaisquer tags HTML. Dessa forma, ele retorna apenas o conteúdo textual sem nenhuma formatação ou marcação HTML. Se houver um parágrafo com texto e tags HTML dentro dele, o **textContent** retornará apenas o texto, ignorando as tags HTML.
- **innerHTML**: O **innerHTML** é outra propriedade que permite acessar e modificar o conteúdo de um elemento HTML, mas ao contrário do **innerText**, ele retorna todo o conteúdo HTML dentro desse elemento, incluindo qualquer marcação HTML presente. Isso significa que além do texto, você também tem acesso às tags HTML internas. Por exemplo, se você tem um parágrafo **<p>** com texto dentro dele e uma tag de negrito **** dentro do parágrafo, o **innerHTML** retornará o texto do parágrafo junto com a tag ****.

Em resumo, enquanto **innerText** e **textContent** fornecem apenas o texto visível do elemento, **innerHTML** fornece todo o conteúdo HTML dentro dele. É importante usar cada um deles com cuidado, especialmente **innerHTML**, pois ao modificar o conteúdo de um elemento usando essa propriedade, você está essencialmente injetando HTML diretamente na página, o que pode representar um risco de segurança se o conteúdo não for confiável.

Dentro do nosso arquivo "index.html", além de modificar o elemento button existente, adicionaremos mais dois elementos botão.

```
11 <button onclick="criarElementoInnerText()">Adicionar Elemento com InnerText</button>
12 <button onclick="criarElementoInnerHTML()">Adicionar Elemento com innerHTML</button>
13 <button onclick="criarElementoTextContent()">Adicionar Elemento com TextContent</button>
...
```

Vamos alterar a nossa função **criarElemento** do nosso arquivo "script.js"; vamos renomeá-la para **criarElementoInnerText** e também adicionar a linha que implementará a utilização do nosso **innerText** dentro do nosso elemento **li**.

```
1 function criarElementoInnerText() {
2   const ul = document.getElementById("lista-de-elementos");
3
4   const novaLi = document.createElement("li");
5   novaLi.innerText = "Novo input com InnerText";
6
7   const novoInput = document.createElement("input");
8
9   novaLi.appendChild(novoInput);
10  ul.appendChild(novaLi);
11}
12}
```

Criando novos Elementos no DOM com Javascript

- Adicionar Elemento com InnerText
- Adicionar Elemento com innerHTML
- Adicionar Elemento com TextContent

- Novo input com InnerText

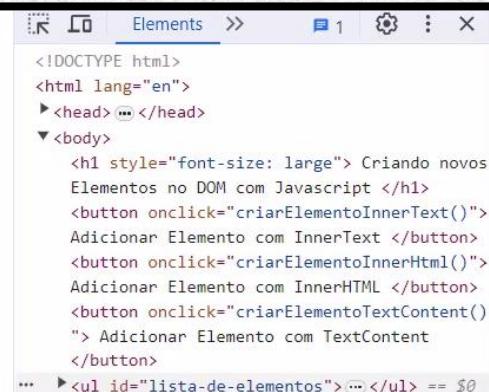


Agora vamos implementar a segunda função, que é a "criarElementoInnerHTML()" que será responsável por armazenar as instruções de criação de elementos utilizando o método innerHTML.

```
function criarElementoInnerHTML() {  
  const ul = document.getElementById("lista-de-elementos");  
  const novaLi = document.createElement("li");  
  
  // console.log("Adicionando novo elemento ao clicar");  
  novaLi.innerHTML = '<input type="text" />';  
  ul.appendChild(novaLi);  
}
```

Criando novos Elementos no DOM com Javascript

- Adicionar Elemento com InnerText
- Adicionar Elemento com InnerHTML
- Adicionar Elemento com TextContent



The screenshot shows the browser's developer tools with the 'Elements' tab selected. The DOM tree on the right side of the tool shows the following structure:

```
<!DOCTYPE html>  
<html lang="en">  
  <head> ... </head>  
  <body>  
    <h1 style="font-size: large"> Criando novos Elementos no DOM com Javascript </h1>  
    <button onclick="criarElementoInnerText()"> Adicionar Elemento com InnerText </button>  
    <button onclick="criarElementoInnerHTML()> Adicionar Elemento com InnerHTML </button>  
    <button onclick="criarElementoTextContent()> Adicionar Elemento com TextContent </button>  
    ... <ul id="lista-de-elementos">... </ul> == $0
```

E a última função que vamos implementar é a "criarElementoTextContent()" que será responsável por armazenar as instruções de criação de elementos utilizando o método textContent.

```
function criarElementoTextContent() {  
    const ul = document.getElementById("lista-de-elementos");  
    const novaLi = document.createElement("li");  
    const novoInput = document.createElement("input");  
  
    novaLi.textContent = "Atribuindo conteúdo TextContent";  
    novaLi.appendChild(novoInput);  
    ul.appendChild(novaLi);  
}
```

Criando novos Elementos no DOM com Javascript

- Adicionar Elemento com InnerText
- Adicionar Elemento com InnerHTML
- Adicionar Elemento com TextContent

- Atribuindo conteúdo TextContent



Ambas as propriedades **innerText** e **textContent** são utilizadas para acessar ou modificar o conteúdo textual de um elemento HTML, mas existem diferenças importantes entre elas:

- **Comportamento em relação à renderização de CSS e conteúdo oculto:**

- **innerText:** Retorna o texto visível de um elemento, levando em consideração o estilo CSS aplicado e qualquer conteúdo oculto por CSS (como elementos com a propriedade **display: none;**).
- **textContent:** Retorna o conteúdo textual completo de um elemento, independentemente do estilo CSS aplicado ou se o conteúdo está visível ou não na página. Isso significa que ele retorna todo o texto, incluindo o texto de elementos ocultos ou comentários HTML.

- **Desempenho:**

- **innerText:** Como leva em consideração a renderização visual, pode ser mais lento em operações que envolvem grandes quantidades de elementos ou mudanças frequentes no conteúdo, já que requer o cálculo do layout.
- **textContent:** Por não considerar a renderização visual, geralmente é mais rápido do que **innerText**, especialmente em operações que envolvem muitos elementos ou mudanças no conteúdo.

- **Suporte e compatibilidade:**

- **innerText:** É mais amplamente suportado em navegadores mais antigos e tem uma compatibilidade melhor do que **textContent**.
- **textContent:** Embora seja mais rápido e preciso em muitos casos, pode não ser totalmente suportado em navegadores mais antigos.

Módulo 9 – Diferença innerText e textContent (2 / 4)

Para observarmos essas diferenças, vamos iniciar o nosso arquivo "index.html" com a seguinte estrutura:

```

<index.html> ...
1   <!DOCTYPE html>
2   <html lang="en">
3     <head>
4       <meta charset="UTF-8" />
5       <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6       <title>InnerText e TextContent</title>
7       <style></style>
8     </head>
9     <body>
10       <h1 style="font-size: large">Diferença de InnerText e TextContent</h1>
11
12       <button onclick="show()">Diferenças</button>
13
14       <div id="elemento-oculto" class="oculto">
15         | Este é um exemplo de diferença entre innerText e TextContent
16       </div>
17     </body>
18   </html>

```

Diferença de InnerText e TextContent

Diferenças

Este é um exemplo de diferença entre
innerText e TextContent

A tag **<style>** dentro do elemento **<head>** de um documento HTML é usada para definir estilos CSS que serão aplicados ao conteúdo HTML desse documento. Aqui está como funciona:

- Localização:** A tag **<style>** é colocada dentro da seção **<head>** do seu documento HTML. O **<head>** contém metadados e informações sobre o documento, enquanto o **<style>** dentro dele especifica como o conteúdo deve ser estilizado.
- Sintaxe:** Dentro da tag **<style>**, você escreve regras CSS. Isso inclui seletores (como classes, IDs ou elementos HTML) e as propriedades de estilo que deseja aplicar a esses seletores.
- Usar CSS dentro da tag **<style>** é útil quando você tem estilos específicos para apenas uma página ou um conjunto limitado de páginas. Se você precisar de estilos que serão compartilhados entre várias páginas, é recomendável usar uma folha de estilo externa (com a extensão .css) e vinculá-la ao seu documento HTML usando a tag **<link>**.

Dentro da tag style, vamos adicionar a nossa classe .oculto e adicionar uma propriedade para ela:

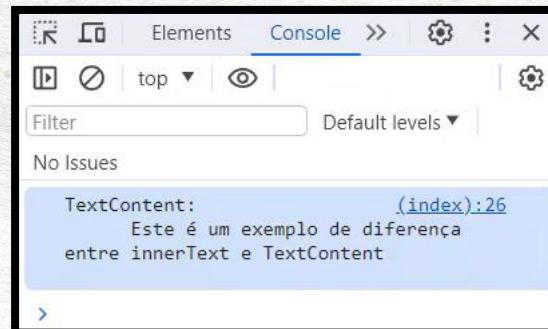
- **visibility: hidden;**: Esta é uma propriedade CSS que usamos para fazer algo desaparecer da tela. Mas, ao contrário de **display: none;**, o espaço que ele ocupava ainda está lá na página, apenas não podemos ver o elemento.

Então, quando alguém usa **.oculto { visibility: hidden; }**, eles estão dizendo ao navegador para tornar invisível um elemento, mas ainda manter o espaço reservado para ele na página. Isso é útil quando você quer esconder algo, mas quer que o layout da página permaneça o mesmo.

```
<style>
  .oculto {
    visibility: hidden;
  }
</style>
```

```
22 <script>
23   function show() {
24     const div = document.getElementById("elemento-oculto");
25
26     console.log("TextContent: ", div.textContent);
27   }
28 </script>
```

Agora vamos adicionar uma tag script e dentro dessa tag vamos implementar a lógica da nossa função "show()", utilizando primeiro o método textContent.



E ao clicar no botão "Diferenças", conseguimos observar que o conteúdo do nosso elemento aparece no nosso console do devTools.

Módulo 9 – Diferença innerText e textContent (4 / 4)

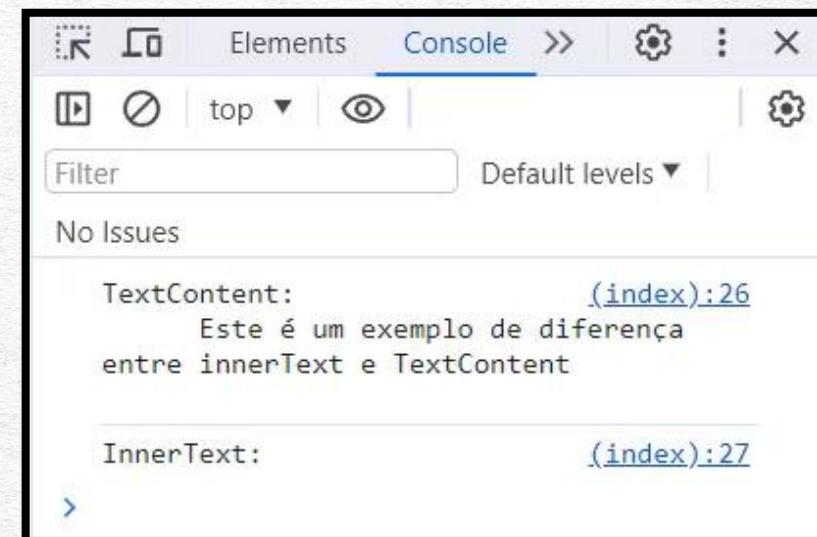
Agora vamos alterar nossa função "show()" e implementar o método innerText, como o código ao lado.

E quando executamos o clique no botão, conseguimos observar a diferença entre esses métodos.

Enquanto o "textContent" consegue exibir o conteúdo dentro do elemento, o innerText não mostra o conteúdo do mesmo elemento, pois ele respeita a regra do CSS da propriedade visibility:hidden. Se não houvesse essa regra na folha de estilo, ambos mostrariam o conteúdo do elemento.

```
<script>
  function show() {
    const div = document.getElementById("elemento-oculto");

    console.log("TextContent: ", div.textContent);
    console.log("InnerText: ", div.innerText);
  }
</script>
```



Nessa aula vamos implementar conceitos para modificar atributos de elementos html, e para iniciar temos o nosso arquivo "index.html" e "style.css" já implementados com a seguinte estrutura:

```
↳ index.html > ...
1  <!DOCTYPE html>
2  <html lang="en">
3  | <head>
4  | | <meta charset="UTF-8" />
5  | | <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6  | | <title>Modificando Atributos HTML</title>
7  | | <link rel="stylesheet" href="style.css" />
8  | </head>
9  <body>
10 | <h1>Modificando Atributos HTML</h1>
11 | <div>
12 | | <button onclick="changeImage()">Trocar Imagem</button>
13 | | 
14 | </div>
15 |
16 | <div>
17 | | <button onclick="changeType()">Mudar tipo de input</button>
18 | | <input type="text" />
19 | </div>
20
21 | <script src="script.js"></script>
22 </body>
23 </html>
```

```
# style.css > ...
1  /* Estilos podem ser adicionados conforme necessário */
2  body {
3  |   font-family: Arial, sans-serif;
4  |   text-align: center;
5  |   margin-top: 50px;
6  }
7
8  img {
9  |   max-width: 300px;
10 |   height: auto;
11 |   display: block;
12 |   margin: 20px auto;
13 }
```

Lembrando que você precisa ter dentro do seu projeto duas imagens quaisquer para poder executar a aula, podendo ser as imagens utilizadas em aulas anteriores.

Os métodos **getAttribute** e **setAttribute** são usados para obter e definir atributos de elementos HTML em JavaScript.

getAttribute(nomeDoAtributo):

- Este método é usado para obter o valor de um atributo específico de um elemento HTML.
- Por exemplo, se você tem um elemento **<div>** com um atributo **id**, você pode usar **getAttribute** para obter o valor desse atributo.

```
let minhaDiv = document.getElementById("minhaDiv");
let divId = minhaDiv.getAttribute("id");
console.log(divId); // Isso irá imprimir o valor do atributo 'id' da div
```

setAttribute(nomeDoAtributo, valor):

- Este método é usado para definir ou alterar o valor de um atributo específico de um elemento HTML.
- Se o atributo já existir, o seu valor será substituído pelo novo valor. Se o atributo não existir, ele será criado.

```
let minhaDiv = document.getElementById("minhaDiv");
minhaDiv.setAttribute("id", "novaDiv");
```

Neste exemplo, o atributo **id** da div com id **minhaDiv** é alterado para **novaDiv**.

Esses métodos são úteis para manipular atributos HTML dinamicamente através do JavaScript, o que é comumente usado em tarefas de manipulação do DOM (Document Object Model).

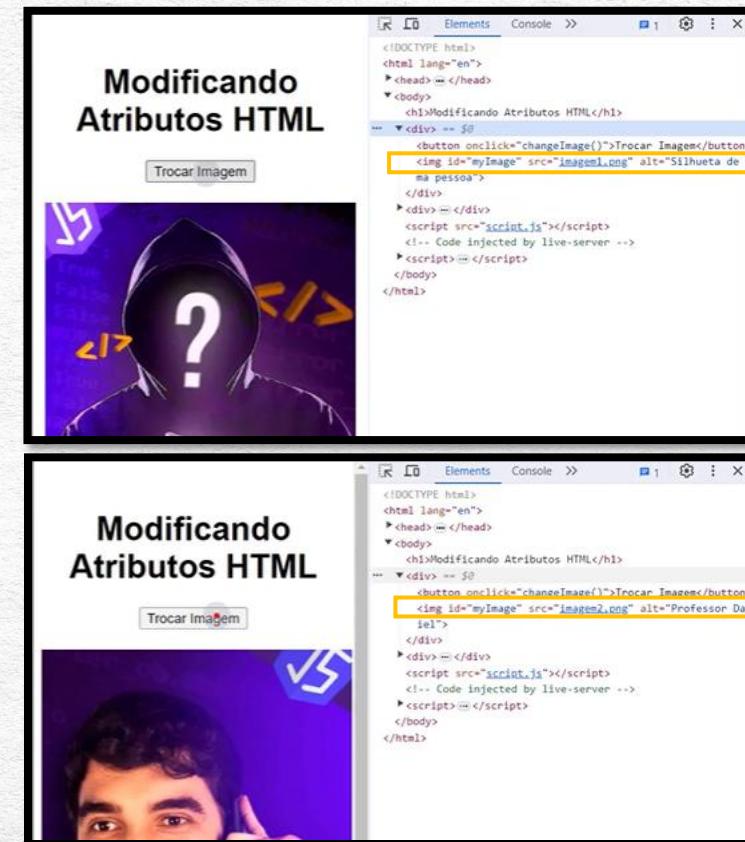


Módulo 9 – Modificando atributos com Javascript (3 / 5)

631

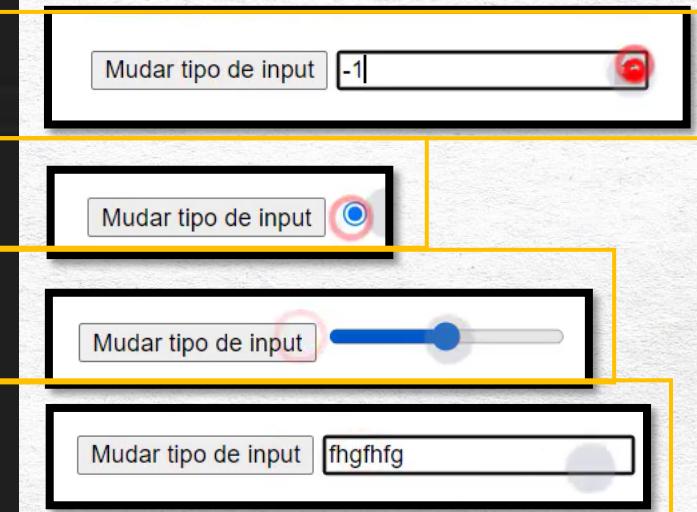
Neste momento, vamos implementar a lógica da nossa função 'changeImage()' dentro do arquivo 'script.js', para que ela seja capaz de alterar a propriedade 'src' do elemento img, com o objetivo de trocar a imagem que está aparecendo no nosso programa.

```
JS script.js > ...
1 // getAttribute - obter o valor do atributo
2 // setAttribute - definir ou modificar o valor
3
4 function changeImage() {
5   const img = document.getElementById("myImage");
6   const currentSrc = img.getAttribute("src");
7
8   if (currentSrc === "imagem1.png") {
9     img.setAttribute("src", "imagem2.png");
10    img.setAttribute("alt", "Professor Daniel");
11  } else {
12    img.setAttribute("src", "imagem1.png");
13    img.setAttribute("alt", "Silhueta de uma pessoa");
14  }
15}
```



Agora vamos implementar a lógica da nossa função "changeType()" dentro do arquivo 'script.js', para que ela seja capaz de alterar a propriedade "type" do elemento input, com o objetivo de trocar o tipo de input que está aparecendo no nosso programa.

```
17 function changeType() {
18     const input = document.getElementsByTagName("input")[0];
19     const currentType = input.getAttribute("type");
20
21     switch (currentType) {
22         case "text":
23             input.setAttribute("type", "number");
24             break;
25         case "number":
26             input.setAttribute("type", "radio");
27             break;
28         case "radio":
29             input.setAttribute("type", "range");
30             break;
31         default:
32             input.setAttribute("type", "text");
33     }
34 }
```



O método **getElementsByName** retorna uma coleção (ou uma lista) de elementos com o nome da tag especificada (neste caso, "input"). Mesmo que saibamos que só existe um elemento **<input>** no documento, o método **getElementsByName** ainda retorna uma coleção de elementos, não o próprio elemento. Portanto, para acessar esse único elemento, precisamos especificar o índice **[0]** para obter o primeiro elemento dessa coleção.

O método **getAttribute** em JavaScript retorna uma string representando o valor do atributo desejado.

```
const currentInput = input.getAttribute("type");
```

Você está obtendo apenas o valor atual do atributo **type** do elemento **<input>**, mas não está vinculado diretamente ao elemento em si. Portanto, ao alterar **currentInput**, você estaria apenas manipulando uma string que representa o valor do atributo **type**, não o próprio atributo do elemento.

```
input.setAttribute("type", "text");
```

Você está diretamente alterando o atributo **type** do elemento **<input>** para o valor "text". Isso altera o estado do elemento no DOM, o que resultará em mudanças visíveis no navegador.

Portanto, enquanto **getAttribute** é útil para obter valores de atributos, **setAttribute** é usado para definir ou modificar esses atributos diretamente no elemento HTML.

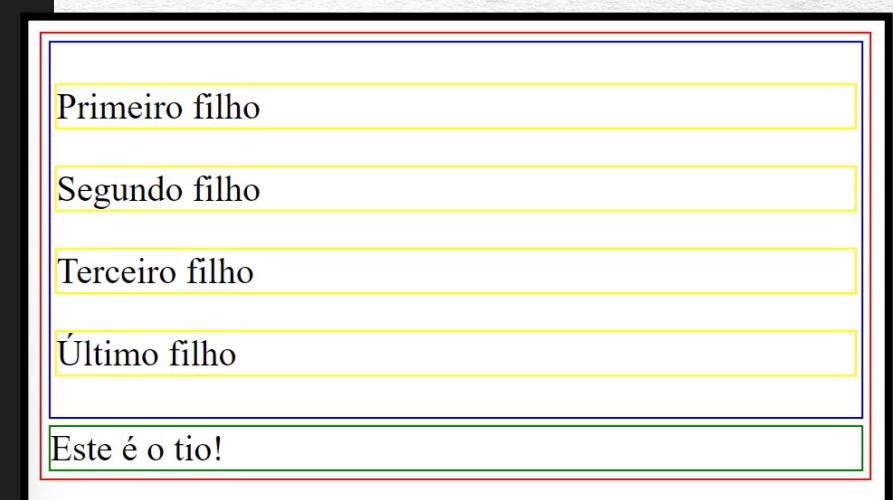
Nesta aula vamos compreender melhor a relação entre os nós da árvore do DOM, e alguns métodos para acessar os elementos através dessas relações. O código inicial do programa é esse abaixo, e vamos implementar com Javascript os métodos para acessar esses elementos.

```

index.html > ...
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="UTF-8">
5    <meta name="viewport" content="width=device-width, initial-scale=1.0">
6    <title>Percorrendo o DOM</title>
7    <link rel="stylesheet" href="./style.css">
8  </head>
9  <body>
10   <div id="avô">
11     <div id="pai">
12       <p>Primeiro filho</p>
13       <p>Segundo filho</p>
14       <p>Terceiro filho</p>
15       <p>Último filho</p>
16     </div>
17
18     <div id="tio">Este é o tio!</div>
19   </div>
20
21   <script>
22     // CÓDIGO JAVASCRIPT
23   </script>
24 </body>
25 </html>
```

```

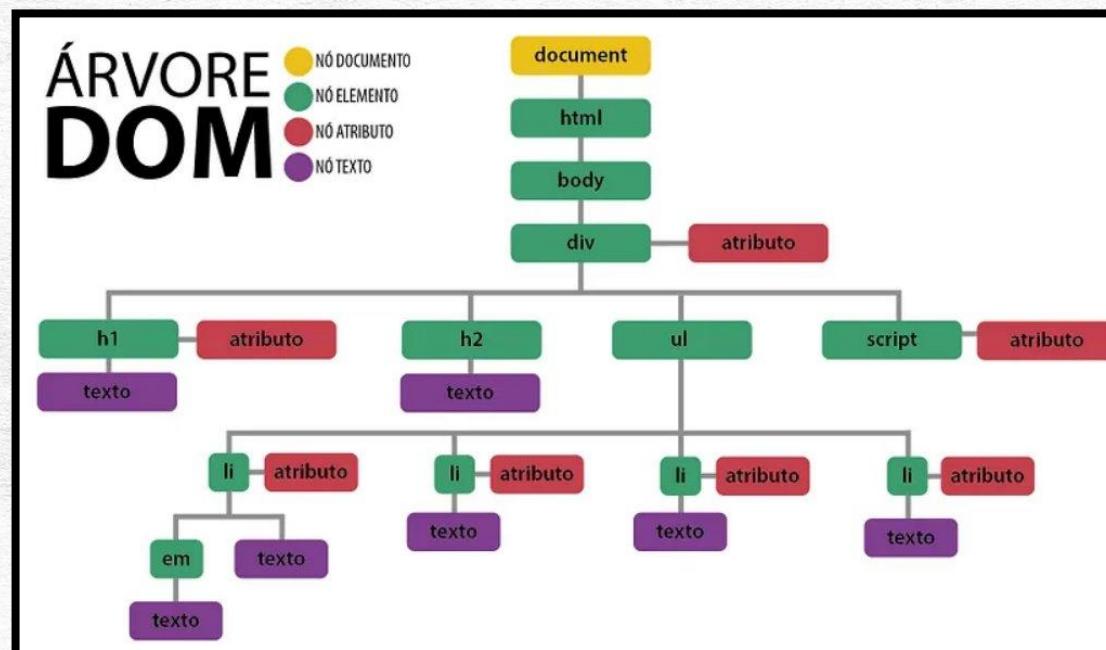
# style.css > ...
1  #avô {
2    border:1px solid red;
3    padding:3px
4  }
5
6  #pai {
7    border:1px solid blue;
8    margin-bottom:3px;
9    padding:2px
10 }
11
12  #tio {
13    border:1px solid green
14  }
15
16  p {
17    border:1px solid yellow
18  }
```



Agora, vamos olhar para a estrutura do código do slide anterior, imagine que seu código HTML é como uma árvore, onde cada elemento é um galho ou uma folha, e nesse caso vamos focar na estrutura que está dentro body.

- **<div id="avô">**: Este é o elemento principal ou "avô". Ele contém dois elementos filhos: "pai" e "tio".
- **<div id="pai">**: Este é um elemento filho do "avô". Ele contém quatro elementos filhos **<p>**, que são parágrafos.
- **<div id="tio">**: Este é outro elemento filho do "avô", ao mesmo nível do elemento "pai". Ele é apenas um bloco de texto.

Então, quando você vê **<div id="avô">**, pense nisso como o topo da árvore, e quando você vê **<div id="pai">** e **<div id="tio">**, pense neles como ramos que saem do topo, e os parágrafos como ramos que saem do "pai". Essa estrutura hierárquica é fundamental para entender como os elementos de uma página web estão organizados e como você pode interagir com eles usando JavaScript



No HTML, os elementos são organizados em uma estrutura hierárquica, onde um elemento pode conter outros elementos dentro de si. Essa relação é fundamental para a estruturação e formatação das páginas web.

Vamos começar com os conceitos básicos:

- **Elemento Pai:** Um elemento pai é aquele que contém outro elemento dentro de si. É o elemento que envolve e engloba outros elementos. Pense nele como o contêiner principal.
- **Elemento Filho:** Um elemento filho é aquele que está contido dentro de outro elemento. Ele é parte do conteúdo do elemento pai. Os elementos filhos podem, por sua vez, conter outros elementos, criando uma estrutura hierárquica.

A relação pai-filho é fundamental para aplicar estilos CSS, manipular elementos usando JavaScript e também para estruturar semanticamente o conteúdo da página.

Por exemplo, se quisermos aplicar um estilo CSS específico apenas ao parágrafo dentro do elemento pai, podemos fazer isso referenciando o parágrafo como um filho do elemento pai:

```
#pai p {  
    color: blue;  
}
```

Este código CSS irá aplicar a cor azul apenas aos parágrafos que são filhos diretos do elemento com o ID "pai".

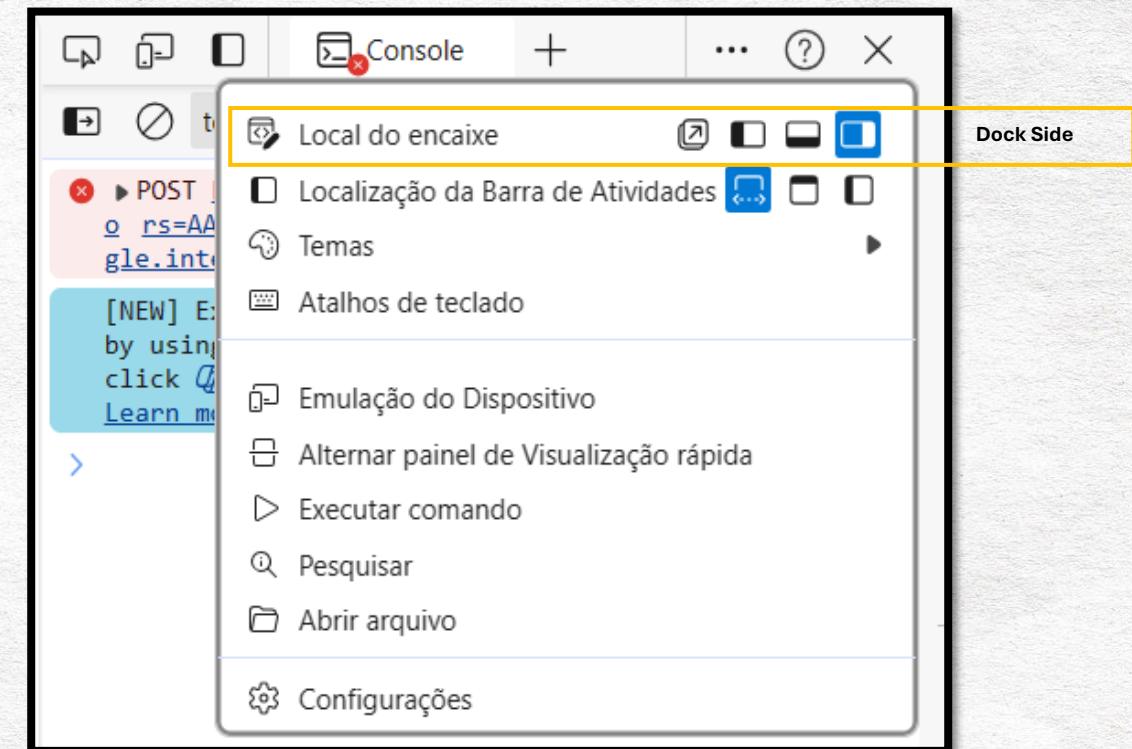
Além disso, a manipulação do DOM (Document Object Model) usando JavaScript frequentemente envolve navegar pela estrutura pai-filho para acessar e modificar elementos específicos.





DevTools - Posicionamento

O "local de encaixe" refere-se à posição onde a janela ou painel de ferramentas, como o devtools, pode ser ancorado ou fixado na interface de usuário de um aplicativo ou ambiente de desenvolvimento.



O DOM (Document Object Model) é uma representação hierárquica dos elementos HTML em uma página da web. Em JavaScript, podemos acessar diferentes partes dessa estrutura usando várias propriedades e métodos. Vamos explorar as propriedades mais comuns que nos permitem navegar pelo DOM:

- **parentNode:**
 - Esta propriedade aponta para o nó pai de um elemento no DOM.
 - Exemplo: Se temos um elemento `<div id="avô">`, o **parentNode** desse elemento será o elemento pai mais próximo, como `<body>`. Se não houver um nó pai, **parentNode** será **null**.
- **childNodes:**
 - Retorna uma coleção de todos os nós filhos de um elemento, incluindo nós de texto, comentários e elementos HTML.
 - Diferença de children: Inclui todos os tipos de nós, não apenas elementos HTML.
- **children:**
 - Similar a **childNodes**, mas retorna apenas os nós filhos que são elementos HTML, excluindo nós de texto e comentários.
- **firstChild:**
 - Retorna o primeiro nó filho de um elemento, incluindo nós de texto e comentários, além de elementos HTML. Se não houver nenhum nó filho, retorna **null**.
- **firstElementChild:**
 - Similar a **firstChild**, mas retorna o primeiro filho que é um elemento HTML, excluindo nós de texto e comentários.

- **lastChild:**

- Retorna o último nó filho de um elemento, incluindo nós de texto e comentários, além de elementos HTML. Se não houver nenhum nó filho, retorna **null**.

- **lastElementChild:**

- Similar a **lastChild**, mas retorna o último filho que é um elemento HTML, excluindo nós de texto e comentários.

- **nextSibling:**

- Retorna o próximo nó irmão de um nó específico no DOM, incluindo nós de texto, comentários e elementos HTML. Se não houver próximo nó irmão, retorna **null**.

- **nextElementSibling:**

- Similar a **nextSibling**, mas retorna o próximo irmão que é um elemento HTML, excluindo nós de texto e comentários.

- **previousSibling:**

- Retorna o nó irmão anterior de um nó específico no DOM, incluindo nós de texto, comentários e elementos HTML. Se não houver nó irmão anterior, retorna **null**.

- **previousElementSibling:**

- Similar a **previousSibling**, mas retorna o irmão anterior que é um elemento HTML, excluindo nós de texto e comentários.

- **nodeType:**

- Propriedade de nós no DOM que retorna um número representando o tipo do nó. O valor 1 corresponde a elementos HTML, o valor 3 corresponde a nós de texto e o valor 8 corresponde a nós de comentário. Essa propriedade é útil para filtrar nós específicos em uma coleção de nós quando você deseja acessar apenas elementos HTML.



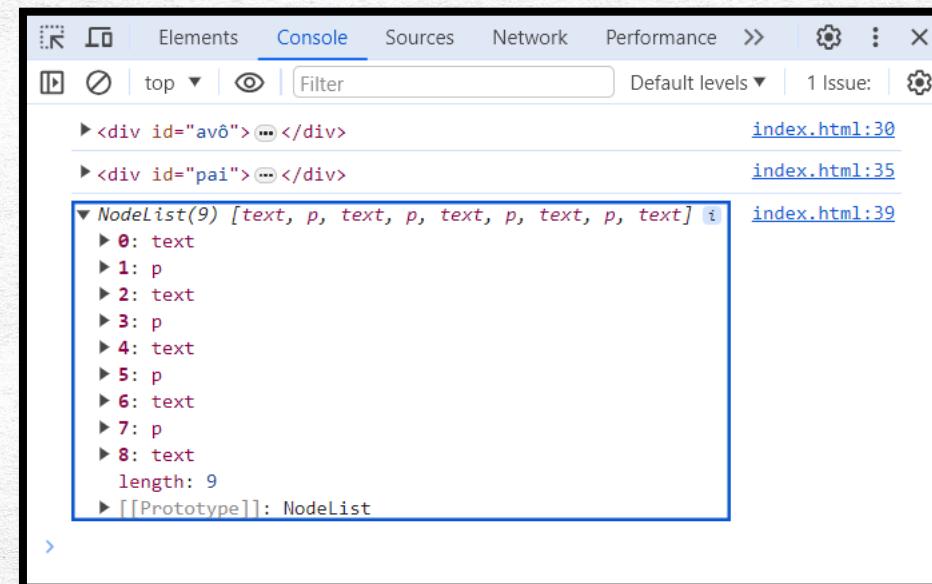
Abaixo vamos observar o retorno de 2 propriedades , como elas podem ser acessadas e seus retornos no console:

```
// obter o elemento pai
let pai = document.getElementById("pai");

// Acessar o parentNode Avô
let avo = pai.parentNode;
console.log(avo);

// Acessar o parentNode Pai pelos filhos
let filhosParagrafo = document.querySelector("p");
let paiAtravesFilhos = filhosParagrafo.parentNode;
console.log(paiAtravesFilhos);

// Acessar os nós filhos usando childNodes
let filhosChildNodes = pai.childNodes;
console.log(filhosChildNodes);
```



Continuando a analisar o retorno de propriedades , como são acessadas e retornadas no console:

```
// Acessar o parentNode Pai pelos filhos
let filhosParagrafo = document.querySelector("p");
let paiAtravesFilhos = filhosParagrafo.parentNode;
console.log(paiAtravesFilhos);

// Acessar os nós filhos usando childNodes
let filhosChildNodes = pai.childNodes;
console.log(filhosChildNodes);

// Acessar os nós filhos usando children
let filhosChildren = pai.children;
console.log(filhosChildren);
```

```
> <div id="pai">@@</div> index.html:35
▼ NodeList(9) [text, p, text, p, text, p, text, p, text] ⓘ index.html:39
  ► 0: text
  ► 1: p
  ► 2: text
  ► 3: p
  ► 4: text
  ► 5: p
  ► 6: text
  ► 7: p
  ► 8: text
  length: 9
  ► [[Prototype]]: NodeList

▼ HTMLCollection(4) [p, p, p, p] ⓘ index.html:43
  ► 0: p
  ► 1: p
  ► 2: p
  ► 3: p
  length: 4
  ► [[Prototype]]: HTMLCollection
```

Continuando a analisar o retorno de propriedades , como são acessadas e retornadas no console:

```
▼ HTMLCollection(4) [p, p, p, p] ⓘ index.html:43
  ► 0: p
  ► 1: p
  ► 2: p
  ► 3: p
    length: 4
  ► [[Prototype]]: HTMLCollection

  ► #text index.html:48
  ► p index.html:49
  ► #text index.html:54
  <p>Último filho</p> index.html:55
```

```
// Acessar os nós filhos usando children
let filhosChildren = pai.children;
console.log(filhosChildren);

// Acessar o primeiro filho usando firstChild
let primeiroFilho = pai.firstChild;
let primeiroFilhoElemento = pai.firstElementChild;
console.log(primeiroFilho);
console.log(primeiroFilhoElemento);

// Acessar o último filho usando lastChild
let ultimoFilho = pai.lastChild;
let ultimoFilhoElemento = pai.lastElementChild;
console.log(ultimoFilho);
console.log(ultimoFilhoElemento);
```

```
// Acessar o próximo irmão usando nextSibling
let segundoFilho = primeiroFilho.nextElementSibling;
console.log(segundoFilho);

// Acessar o irmão anterior usando previousSibling
let terceiroFilho = ultimoFilho.previousElementSibling;
console.log(terceiroFilho);

// Acessar o tio usando nextElementSibling
let tio = pai.nextElementSibling;
console.log(tio);

// Acessar o pai usando previousElementSibling
let paiPrevious = tio.previousElementSibling;
console.log(paiPrevious);
```



E por fim, vamos observar como podemos utilizar outras estruturas para percorrer os elementos:

```
// Acessando apenas os elementos, sem pegar os espaços em brando nodeType
let tioElemento = pai.nextSibling;
console.log(tioElemento)

while (tioElemento.nodeType !== 1) {
    tioElemento = tioElemento.nextSibling;
}
console.log("Elemento tio:", tioElemento);
```

▶ #text	index.html:75
Elemento tio: <div id="tio">Este é o tio!</div>	index.html:80
>	

Módulo 9 – Inserindo e Removendo elementos (1 / 5)

Nessa aula vamos ver como manipular o DOM, inserindo e removendo elementos utilizando Javascript. Para iniciarmos, vamos implementar a estrutura abaixo dentro de um arquivo html e css:

```
index.html > html > head > title
1  <!DOCTYPE html>
2  <html lang="en">
3  | <head>
4  | | <meta charset="UTF-8" />
5  | | <title>Manipulação do DOM</title>
6  | | <link rel="stylesheet" href="style.css" />
7  | </head>
8  <body>
9  | <div id="container">
10 | | <h1>Manipulação do DOM</h1>
11 | | <ul id="list">
12 | | | <li>Item 1</li>
13 | | | <li>Item 2</li>
14 | | | <li>Item 3</li>
15 | | </ul>
16
17 | <button id="addBtn" onclick="adicionarItem()">Adicionar Item</button>
18 | <button id="removeBtn" onclick="removerItem()">Remover Item</button>
19 | </div>
20
21 <script src="script.js"></script>
22 </body>
23 </html>
```

```
# style.css > body
1  body {
2  font-family: Arial, sans-serif;
3  margin: 0;
4  padding: 0;
5  }
6
7  #container {
8  max-width: 600px;
9  margin: 20px auto;
10 padding: 20px;
11 border: 1px solid #ccc;
12 border-radius: 5px;
13 }
14
15 ul {
16 list-style: none;
17 padding: 0;
18 }
19
20 button {
21 margin-top: 10px;
22 padding: 8px 12px;
23 cursor: pointer;
24 background-color: #007bff;
25 color: #fff;
26 border: none;
27 border-radius: 3px;
28 }
29
30 button:hover {
31 background-color: #0056b3;
32 }
33
34 button:focus {
35 outline: none;
36 }
```



A **manipulação do Document Object Model (DOM)** é um aspecto essencial do desenvolvimento web moderno, especialmente quando se trata de interatividade e dinamismo nas páginas da web. O DOM serve como uma representação estruturada em árvore de um documento HTML, permitindo que os desenvolvedores acessem, alterem e atualizem os elementos e conteúdos da página em tempo real.

Ao carregar uma página da web em um navegador, o navegador constrói uma estrutura de árvore de objetos que representa todos os elementos HTML presentes na página. Essa estrutura, conhecida como DOM, é manipulável através de JavaScript, o que permite aos desenvolvedores modificar o conteúdo, estilo e comportamento dos elementos da página de forma dinâmica.

A manipulação do DOM com JavaScript envolve a seleção de elementos específicos, a modificação de seus atributos e conteúdos, a adição ou remoção de elementos e a resposta a eventos do usuário, como cliques e teclas pressionadas. Essa capacidade de interação direta com a estrutura da página é fundamental para a criação de experiências web ricas e responsivas.

Neste contexto, exploraremos os conceitos fundamentais da inserir e remover elementos do DOM com JavaScript, demonstrando técnicas e práticas comuns usadas pelos desenvolvedores para criar aplicações web dinâmicas e interativas.

Então vamos implementar duas lógicas dentro de um arquivo script.js, contendo duas funções: **adicionarItem()** e **removerItem()**.

Esse código JavaScript tem o propósito de adicionar um novo item à lista no DOM (Document Object Model) de uma página web. Vamos analisar os métodos e propriedades utilizados, além de como eles se relacionam com a manipulação do DOM:

```
JS script.js > ...
1  function adicionarItem() {
2      const lista = document.getElementById("list");
3      const novaLi = document.createElement("li");
4
5      novaLi.textContent = "Novo Item: ";
6
7      lista.appendChild(novaLi);
8  }
```

- **getElementById("list")**: Este método é usado para obter uma referência ao elemento HTML com o ID "list". Presumivelmente, este é o elemento **** que representa a lista onde desejamos adicionar um novo item. Ele retorna uma referência ao elemento encontrado ou **null** se nenhum elemento corresponder ao ID especificado.
- **createElement("li")**: Este método cria um novo elemento HTML ****. No contexto deste código, estamos criando um novo item de lista que será adicionado à lista existente.
- **textContent = "Novo Item:"**: Esta propriedade define o texto dentro do elemento **** que acabamos de criar. Ele define o conteúdo de texto para "Novo Item: ".
- **appendChild(novaLi)**: Este método é usado para adicionar um nó filho ao final da lista de nós de um elemento pai. Aqui, estamos adicionando o novo elemento **** que criamos anteriormente como um novo item na lista.

Juntando isso com a manipulação do DOM, vemos que estamos basicamente criando um novo elemento de lista (****) e definindo seu conteúdo de texto como "Novo Item: ". Em seguida, adicionamos esse novo elemento à lista existente no DOM. Isso é uma forma de dinamicamente alterar o conteúdo de uma página web sem precisar recarregá-la.



Esse código JavaScript tem como objetivo remover o último item de uma lista no DOM. Vamos analisar suas partes e como elas se relacionam com a manipulação do DOM:

- **const lista = document.getElementById("list");**: Esta linha de código obtém uma referência ao elemento HTML com o ID "list". Presumivelmente, este é o elemento **** que representa a lista da qual desejamos remover um item. Essa referência é armazenada na variável **lista**.
- **function removerItem() {...}**: Este trecho define uma função chamada **removerItem()**. Esta função será chamada quando quisermos remover o último item da lista.
- **const ultimoItem = lista.lastElementChild;**: Aqui, estamos acessando o último elemento filho do elemento **** (ou seja, o último item da lista). Isso é feito usando a propriedade **lastElementChild** da variável **lista**, que é a referência ao elemento **** obtida anteriormente.
- **lista.removeChild(ultimoItem);**: Esta linha de código remove o elemento filho especificado (**ultimoItem**) do elemento pai (**lista**). Ou seja, estamos removendo o último item da lista.

Quando a função **removerItem()** é chamada, ela primeiro identifica o último item da lista, armazenando-o na variável **ultimoItem**. Em seguida, remove esse último item da lista, utilizando o método **removeChild()** no elemento **** que representa a lista. Isso resulta na remoção do último item da lista no DOM.

```
10 const lista = document.getElementById("list");
11
12 function removerItem() {
13     // Remover o último item da lista
14     const ultimoItem = lista.lastElementChild;
15
16     lista.removeChild(ultimoItem);
17 }
18
```

Este trecho de código JavaScript também tem o propósito de remover um item de uma lista no DOM, mas desta vez remove o primeiro item da lista. Vamos analisar suas partes:

```
10 const lista = document.getElementById("list");
11
12 function removerItem() {
13     // Remover o primeiro item da lista
14     const primeiroItem = lista.firstElementChild;
15
16     lista.removeChild(primeiroItem);
17 }
```

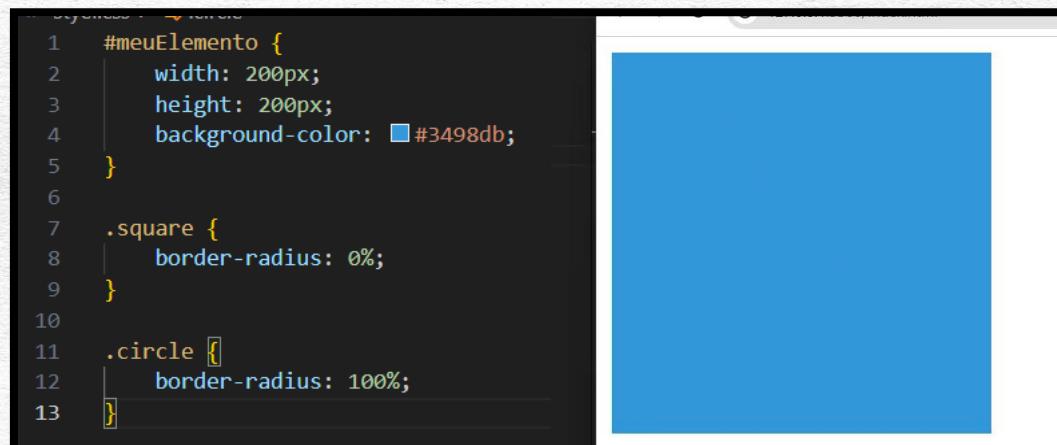
- **function removerItem() {...}**: Assim como no código anterior, esta linha define uma função chamada **removerItem()**, que será chamada quando quisermos remover o primeiro item da lista.
- **const primeiroItem = lista.firstElementChild;**: Aqui, estamos acessando o primeiro elemento filho do elemento **** (ou seja, o primeiro item da lista). Isso é feito utilizando a propriedade **firstElementChild** da variável **lista**, que é uma referência ao elemento ****.
- **lista.removeChild(primeiroItem);**: Esta linha de código remove o elemento filho especificado (**primeiroItem**) do elemento pai (**lista**). Assim, estamos removendo o primeiro item da lista.

Em resumo, quando a função **removerItem()** é chamada, ela identifica o primeiro item da lista, armazenando-o na variável **primeiroItem**, e depois o remove da lista utilizando o método **removeChild()** no elemento **** que representa a lista. Isso resulta na remoção do primeiro item da lista no DOM.

Vamos construir uma modelo básico de uma arquivo de HTML, adicionando na tag title o título da aba, Manipulação do CSS com Javascript. Crie um link para um arquivo externo de estilo CSS chamado "style.css", que será aplicado à página HTML para definir o layout e a aparência. E adicione dentro do <body> um elemento do tipo <div> com ID "meuElemento" e classe "square", como no código abaixo:

```
↳ index.html > ↳ html
1   <!DOCTYPE html>
2   <html lang="en">
3   <head>
4     <meta charset="UTF-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <title>Manipulação do CSS com Javascript</title>
7     <link rel="stylesheet" href="style.css">
8   </head>
9   <body>
10    <div id="meuElemento" class="square"></div>
11  </body>
12 </html>
```

Inicie também o arquivo style.css com a seguinte configuração de estilo:



```
9  <body>
10 <div id="meuElemento" class="square"></div>
11
12 <button onclick="tornarCirculo()">Círculo</button>
13 <button onclick="tornarQuadrado()">Quadrado</button>
14 <button onclick="alternarFormato()">Alternar Formato</button>
15 <button onclick="verificarClasse()">Verificar Classe</button>
16
17 <script src="script.js"></script>
18 </body>
```

O próximo passo é criar quatro botões. Cada botão tem um evento **onclick** associado a uma função JavaScript diferente. Aqui está o que cada botão faz:

- O primeiro botão tem o texto "Círculo" e ao ser clicado, chama a função **tornarCirculo()**.
- O segundo botão tem o texto "Quadrado" e ao ser clicado, chama a função **tornarQuadrado()**.
- O terceiro botão tem o texto "Alternar Formato" e ao ser clicado, chama a função **alternarFormato()**.
- O quarto botão tem o texto "Verificar Classe" e ao ser clicado, chama a função **verificarClasse()**.

Além disso, o código inclui um script externo **script.js**, que deve conter as definições das funções **tornarCirculo()**, **tornarQuadrado()**, **alternarFormato()** e **verificarClasse()**.

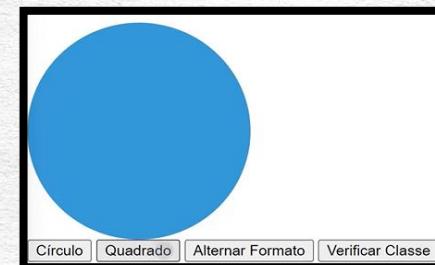
A função chamada **tornarCirculo**, será responsável por tornar um elemento HTML com o ID "meuElemento" em um círculo.

- **const meuElemento = document.getElementById("meuElemento");**: Esta linha está pegando um elemento HTML com o ID "meuElemento" e armazenando-o na variável **meuElemento**.
- **function tornarCirculo() {**: Aqui começa a definição da função **tornarCirculo**.
- **meuElemento.classList.add("circle")**: Esta linha adiciona a classe CSS "circle" ao elemento armazenado em **meuElemento**, tornando-o redondo.
- **meuElemento.classList.remove("square")**: Esta linha remove a classe CSS "square" do elemento, garantindo que ele não seja mais quadrado.

classList, **add** e **remove** são propriedades e métodos que trabalham com as classes CSS de um elemento HTML.

- **classList**: **classList** é uma propriedade de elementos HTML que representa a lista de classes associadas a esse elemento. Permite acessar métodos para adicionar, remover e verificar a presença de classes.
- **add(className)**: **add** é um método da propriedade **classList** que permite adicionar uma classe CSS a um elemento HTML. Você fornece o nome da classe como argumento para este método.
- **remove(className)**: **remove** é um método da propriedade **classList** que permite remover uma classe CSS de um elemento HTML. Você fornece o nome da classe a ser removida como argumento para este método.

```
JS script.js > ...
1 const meuElemento = document.getElementById("meuElemento");
2
3 function tornarCirculo() {
4     meuElemento.classList.add("circle");
5     meuElemento.classList.remove("square");
6 }
```



O código abaixo JavaScript define a função chamada **tornarQuadrado**, que é responsável por transformar um elemento HTML com o ID "meuElemento" em um quadrado.

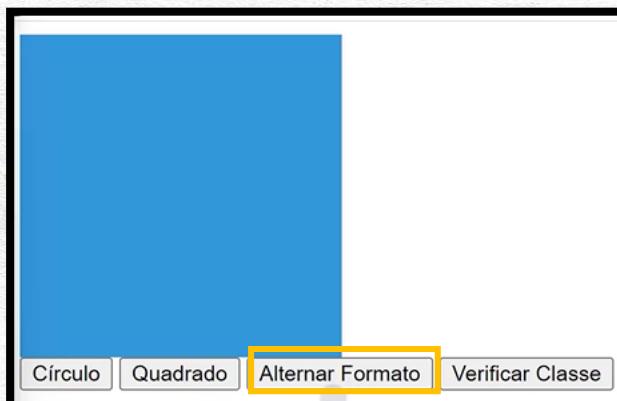
```
8  function tornarQuadrado() {  
9      meuElemento.classList.add("square");  
10     meuElemento.classList.remove("circle");  
11 }
```

- **meuElemento.classList.add("square");**: Esta linha adiciona a classe CSS "square" ao elemento HTML armazenado em **meuElemento**, fazendo com que ele assuma as propriedades de estilo de um quadrado.
- **meuElemento.classList.remove("circle");**: Esta linha remove a classe CSS "circle" do elemento HTML armazenado em **meuElemento**, garantindo que ele não mantenha características de círculo.

A próxima função que vamos criar serve para alternar entre duas classes CSS de um elemento HTML específico. Vamos entender o que ela faz:

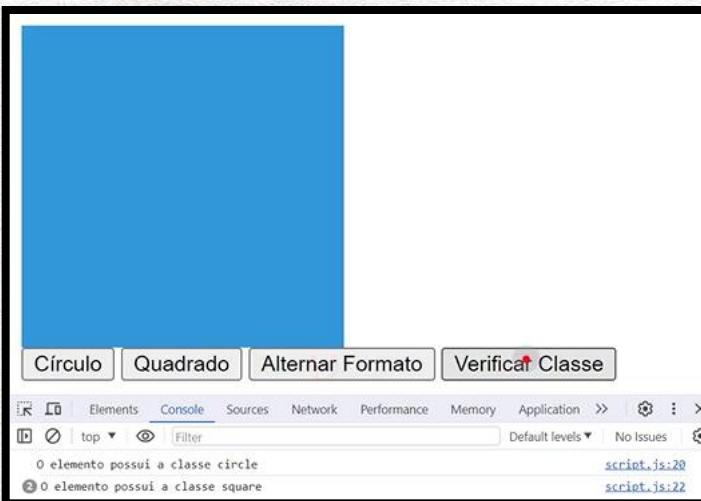
- **meuElemento.classList.toggle("circle");**: Esta linha usa o método **.toggle()** da propriedade **classList** de **meuElemento**. **classList** é uma propriedade que retorna uma coleção de todas as classes de um elemento. O método **.toggle()** adiciona a classe especificada se ela não estiver presente e a remove se estiver presente. No caso, está alternando a presença da classe "círculo" no **meuElemento**.
- **meuElemento.classList.toggle("square");**: Esta linha é similar à anterior, mas está alternando a presença da classe "quadrado" no **meuElemento**

```
14  ↴ function alternarFormato() {  
15      |     meuElemento.classList.toggle("circle");  
16      |     meuElemento.classList.toggle("square")  
17  }
```



a função chamada **verificarClasse()** verifica se um elemento HTML possui a classe "circle" aplicada a ele. Aqui está uma explicação passo a passo do que ela faz:

- **meuElemento.classList.contains("circle")**: Esta linha verifica se o elemento HTML referenciado pela variável **meuElemento** possui a classe "circle". A função **classList.contains()** retorna true se a classe especificada estiver presente no elemento, caso contrário, retorna false.
- Se o elemento possuir a classe "circle", a função exibirá a mensagem "O elemento possui a classe circle" no console.
- Se o elemento não possuir a classe "circle", a função exibirá a mensagem "O elemento possui a classe square" no console.

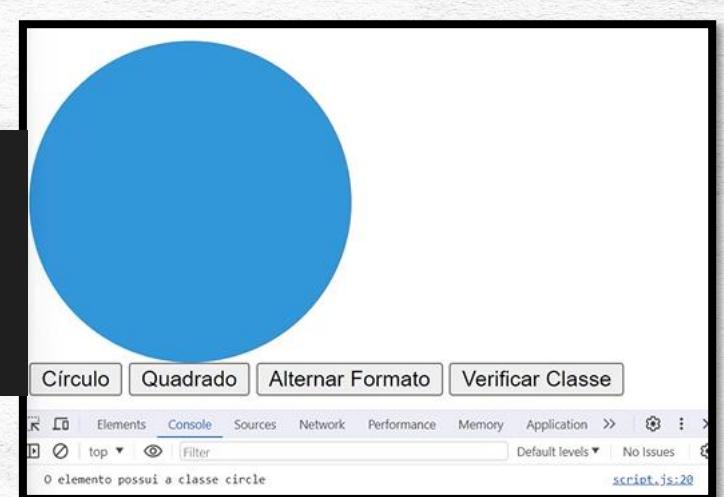


A screenshot of a browser's developer tools console. At the top, there are four buttons: 'Círculo', 'Quadrado', 'Alternar Formato', and 'Verificar Classe'. The 'Círculo' button is highlighted. Below the buttons is a code editor window containing the same JavaScript function as the previous screenshot.

```
19  function verificarClasse() {  
20      if(meuElemento.classList.contains("circle")) {  
21          console.log("O elemento possui a classe circle")  
22      } else {  
23          console.log("O elemento possui a classe square");  
24      }  
25  }
```

The console output shows one entry:

- script.js:20 0 elemento possui a classe circle



```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>Eventos Javascript</title>
7   </head>
8   <body>
9     <h1>Eventos com Javascript</h1>
10
11    <button id="meuBotao" onclick="clicouBotao()">Clique Aqui!</button>
12
13    <script>
14      function clicouBotao() {
15        console.log("Clicou no botão");
16      }
17    </script>
18  </body>
19 </html>
```

Até o momento, contamos com isso para tornar nossos projetos mais robustos e eficientes.

Eventos com Javascript

Clique Aqui!



- **Eventos em JavaScript:** Em JavaScript, eventos são ações que ocorrem no navegador ou no documento que está sendo exibido, como cliques de mouse, pressionamentos de teclas, carregamento de uma página, entre outros. Quando um evento é acionado, ele pode desencadear a execução de código JavaScript, permitindo interatividade e dinamismo em páginas da web.
Os eventos são fundamentais para criar aplicações web interativas e responsivas, pois permitem que os desenvolvedores respondam a ações do usuário e atualizem a interface do usuário em conformidade.
- **Ouvintes de Eventos em JavaScript:** Ouvintes de eventos (event listeners) são mecanismos em JavaScript que aguardam a ocorrência de um evento específico e respondem a ele executando determinado código.

Basicamente, um ouvinte de eventos está "escutando" um evento e, quando esse evento é disparado (como um clique de mouse), o código associado ao ouvinte é executado. Isso permite que os desenvolvedores respondam dinamicamente às ações do usuário. Os ouvintes de eventos são uma parte central da programação de eventos em JavaScript e são amplamente utilizados para adicionar comportamento interativo às páginas da web. Eles são uma maneira eficaz de separar o comportamento do código HTML, tornando o código mais organizado e mantível.

Ao entender esses conceitos, os desenvolvedores podem criar aplicações web mais interativas e responsivas, tornando a experiência do usuário mais agradável e envolvente.

Os métodos **addEventListener** e **removeEventListener** são usados para adicionar e remover ouvintes de eventos em elementos HTML, respectivamente.

- **addEventListener:** Este método é usado para anexar um ouvinte de eventos a um elemento específico no DOM. Ele recebe três argumentos: o tipo de evento a ser ouvido (como "click" ou "keyup"), a função que será executada quando o evento ocorrer e um opcional parâmetro booleano que indica se o ouvinte deve capturar o evento durante a fase de propagação.
- **removeEventListener:** Este método é usado para remover um ouvinte de eventos previamente adicionado a um elemento. Ele requer os mesmos argumentos que o **addEventListener**: o tipo de evento, a função de ouvinte e, opcionalmente, um parâmetro booleano para indicar a fase de propagação.



No exemplo abaixo:

- Primeiro, definimos uma função chamada **clicouBotao**, que simplesmente registra uma mensagem no console quando o botão é clicado.
- Em seguida, selecionamos o elemento HTML com o id "**meuBotao**" usando **document.getElementById("meuBotao")**. Isso retorna uma referência para o elemento botão no DOM.
- Depois, usamos o método **addEventListener** no elemento botão para adicionar um ouvinte de evento. Este método recebe dois argumentos:
 - O primeiro argumento é uma string que especifica o tipo de evento a ser ouvido, neste caso, "click", indicando que queremos detectar cliques no botão.
 - O segundo argumento é a função que será chamada quando o evento ocorrer, neste caso, **clicouBotao**. Note que a função é passada sem os parênteses, pois queremos que ela seja referenciada como um objeto de função e não que seja executada imediatamente.

Essa é a sintaxe básica do **addEventListener** para adicionar um ouvinte de evento a um elemento no DOM. Quando o botão é clicado, a função **clicouBotao** será executada, registrando a mensagem "Clicou no botão" no console.

```
function clicouBotao() {
    console.log("Clicou no botão");
}

// Adicionar um manipulador de evento do clique botão
let botao = document.getElementById("meuBotao");
botao.addEventListener("click", clicouBotao);
```



Vou explicar a sintaxe do **removeEventListener** usando o código abaixo:

```
// Remover o manipulador após 5 segundos
setTimeout(function () {
    botao.removeEventListener("click", clicouBotao);
    console.log("Manipulador de evento foi removido após 5 segundos");
}, 5000);
```

Usamos **setTimeout** para agendar uma função para ser executada após 5 segundos. Esta função será chamada automaticamente após o intervalo especificado.

Dentro da função de **setTimeout**, chamamos **removeEventListener** no elemento botão para remover o ouvinte de evento. Este método recebe os mesmos dois argumentos que o **addEventListener**:

- O primeiro argumento é a string que especifica o tipo de evento que deve ser removido, neste caso, "click".
- O segundo argumento é a referência para a função que foi originalmente passada para **addEventListener**, ou seja, **clicouBotao**.

Após a execução deste código, o ouvinte de evento associado ao clique do botão será removido após 5 segundos, conforme especificado pelo **setTimeout**. Isso significa que a função **clicouBotao** não será mais chamada quando o botão for clicado depois desse intervalo de tempo. Além disso, a mensagem "Manipulador de evento foi removido após 5 segundos" será registrada no console para indicar que o ouvinte foi removido com sucesso.

Vou explicar as diferenças entre os atributos de eventos como **onclick** e a abordagem com **addEventListener**:

Atributos de eventos (como onclick):

- Os atributos de eventos são maneiras antigas e mais simples de lidar com eventos em JavaScript.
- Eles são adicionados diretamente a elementos HTML como atributos, por exemplo, `<button onclick="minhaFuncao()">Clique aqui</button>`.
- A principal limitação é que só pode ser associado um único manipulador de evento a cada evento específico. Se você atribuir um novo manipulador de evento ao mesmo evento, ele substituirá o anterior.
- Eles misturam a lógica do comportamento do HTML com a lógica do JavaScript, o que pode tornar o código menos organizado e mais difícil de manter em aplicativos complexos.
- Eles têm escopo global, o que significa que a função associada ao evento deve ser acessível globalmente ou definida inline.

addEventListener:

- **addEventListener** é uma abordagem mais moderna e flexível para lidar com eventos em JavaScript.
- Ele permite adicionar vários ouvintes de eventos para o mesmo tipo de evento em um elemento, sem substituir os existentes.
- Os ouvintes de eventos são adicionados e removidos de forma programática, o que oferece maior controle sobre o comportamento do evento.
- Ele separa claramente o comportamento do HTML e do JavaScript, resultando em um código mais organizado e modular.
- Ele não tem a limitação de escopo global; os manipuladores de eventos podem ser funções locais, globais ou até mesmo funções anônimas.

Em resumo, enquanto os atributos de eventos são mais simples e diretos para aplicativos simples, **addEventListener** é preferível para aplicativos mais complexos, nos quais há necessidade de associar múltiplos ouvintes de eventos, manter o código mais organizado e separar claramente o comportamento do HTML do comportamento do JavaScript.

Módulo 9 – Atributos vs. AddEventListener() (2 / 3)

Handler

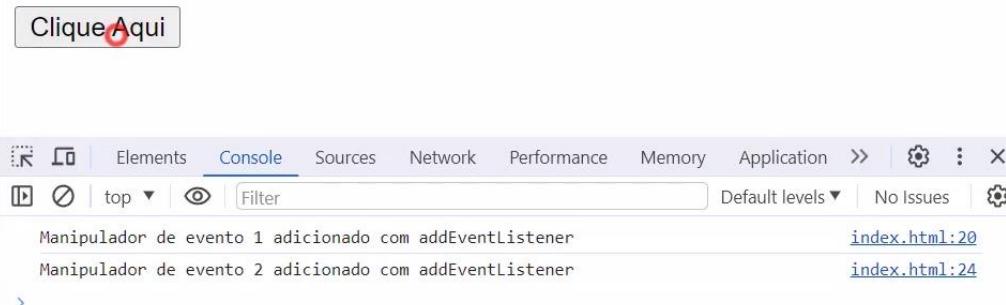
Um "handler" (manipulador) em programação geralmente se refere a uma função ou um bloco de código responsável por lidar com eventos ou exceções.

Abaixo temos um exemplos de como adicionar mais de um manipulador com addEventListener:

```

index.html > html > body > script
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="UTF-8" />
5    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6    <title>Atributo vs. addEventListener</title>
7  </head>
8  <body>
9    <h1>Diferenças entre atributos e addEventListener</h1>
10   <button id="meuBotao">Clique Aqui</button>
11
12   <script>
13     // manipulador de evento = handler
14
15     const botao = document.getElementById("meuBotao");
16
17     // utilizar o addEventListener para adicionar dois manipuladores de eventos de
18     // botao.addEventListener("click", function (evento) {
19     //   console.log("Manipulador de evento 1 adicionado com addEventListener");
20     // });
21
22
23     botao.addEventListener("click", function (evento) {
24       console.log("Manipulador de evento 2 adicionado com addEventListener");
25     });
26   </script>
27 </body>
28 </html>
```

Diferenças entre atributos e addEventListener



O exemplo abaixo mostra como os eventos em atributos atualizam suas funções e não conseguem possuir dois manipuladores ao mesmo tempo.

```
// Utilizar o atributo onclick para adicionar dois manipuladores de eventos
botao.onclick = function (evento) {
    console.log("Manipulador de evento 1 adicionado com onclick");
};

botao.onclick = function (evento) {
    console.log("Manipulador de evento 2 adicionado com onclick");
};
</script>
```

Diferenças entre atributos e addEventListener

Clique Aqui

Log Message	File	Line Number
Manipulador de evento 1 adicionado com addEventListener	index.html	20
Manipulador de evento 2 adicionado com addEventListener	index.html	24
Manipulador de evento 2 adicionado com onclick	index.html	33

Módulo 9 – Prevent Default com addEventListener: Controle de eventos simplificado (1 / 2)

A tag `<a>` é uma das tags mais importantes em HTML e é usada para criar links entre páginas da web ou para seções diferentes da mesma página. Aqui está uma explicação detalhada:

```
<a href="https://www.example.com">Clique aqui para visitar o site de exemplo</a>
```

- **href**: Este é um atributo obrigatório da tag `<a>`. Ele especifica o destino do link, ou seja, para onde o usuário será direcionado quando clicar no link. O valor do atributo **href** é a URL do destino.
- Texto do link: Este é o texto visível que o usuário vê na página da web. É colocado entre a tag de abertura `<a>` e a tag de fechamento ``.

E nesta aula vamos iniciar com o arquivo index.html com a seguinte estrutura:

```
<index.html> ...
1   <!DOCTYPE html>
2   <html lang="en">
3     <head>
4       <meta charset="UTF-8" />
5       <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6       <title>Prevent Default - addEventListener()</title>
7     </head>
8     <body>
9       <h1>Prevent Default</h1>
10
11      <a
12        href="https://dlp.hashtagtreinamentos.com/javascript/intensivao/inscricao-org"
13        id="meuLink"
14      >
15        Clique aqui para ir até a página do Intesivão Javascript</a>
16      >
17    </body>
18  </html>
```



Módulo 9 – Prevent Default com addEventListener: Controle de eventos simplificado (2 / 2)

Quando você adiciona um event listener a um elemento HTML usando **addEventListener**, está essencialmente dizendo ao navegador para ficar atento a um determinado evento (como um clique do mouse, por exemplo) nesse elemento e executar uma função específica quando esse evento ocorrer.

No entanto, em certos casos, você pode querer evitar que o comportamento padrão associado a esse evento ocorra. Por exemplo, ao lidar com um formulário HTML e você deseja validar os dados antes de enviar o formulário, você pode querer interromper o envio do formulário se a validação falhar.

É aí que entra o **preventDefault()**. Este é um método disponível no objeto de evento (geralmente passado como argumento para a função de retorno de chamada do **addEventListener**). Quando você chama **event.preventDefault()**, você está instruindo o navegador a não executar o comportamento padrão associado ao evento que está ocorrendo.

Com a tag **a**, que é usada para criar links em HTML, o uso do **preventDefault()** pode ser útil em situações em que você deseja personalizar o comportamento padrão do navegador ao clicar em um link.

```

18 <script>
19   // Seleciona o elemento link (ancora)
20   const link = document.getElementById("meuLink");
21
22   // Adicionar um evento de clique ao link
23   meuLink.addEventListener("click", function (event) {
24     event.preventDefault();
25
26     console.log("o comportamento padrão do link foi previnido");
27   });
28 </script>
```



Eventos e Interatividade

Em desenvolvimento web centrado em formulários, o JavaScript desempenha um papel fundamental na criação de experiências interativas. Os eventos são disparadores de ações que ocorrem no navegador quando o usuário interage com os elementos do formulário. Alguns dos eventos mais relevantes incluem "submit", "change", "focus" e "blur". Quando o usuário clica em um botão de envio ("submit"), preenche um campo ("change"), foca em um elemento ("focus") ou sai dele ("blur"), esses eventos desencadeiam respostas específicas em JavaScript.

Acesso e Manipulação de Dados

Para acessar e manipular os dados do formulário de forma dinâmica, o JavaScript utiliza o Document Object Model (DOM). O DOM permite que os desenvolvedores acessem e modifiquem os elementos do formulário em tempo real. Com o DOM, é possível adicionar novos elementos, remover itens existentes e atualizar o conteúdo dos campos do formulário conforme necessário. Essa manipulação direta dos elementos do formulário possibilita uma validação e garantia de dados mais eficazes.

As propriedades dos elementos de formulário são essenciais para capturar e processar dados. Ao acessar essas propriedades por meio do DOM, os desenvolvedores podem validar entradas, garantir que campos obrigatórios sejam preenchidos e comparar valores para garantir consistência nos dados fornecidos pelo usuário.

Validação e Garantia de Dados Precisos

A validação de dados em formulários é crítica para garantir que as informações submetidas sejam precisas e seguras. Com JavaScript, é possível verificar campos obrigatórios para garantir que estejam preenchidos e comparar valores para assegurar a consistência dos dados. Por exemplo, ao solicitar uma senha, é fundamental comparar a senha fornecida com sua confirmação para garantir que sejam idênticas.

A comparação de valores também é essencial para garantir a integridade dos dados. Isso pode incluir verificar se um número está dentro de um intervalo específico ou se uma data está no formato correto. Uma validação robusta dos dados não apenas previne erros, mas também protege contra vulnerabilidades, como injeção de SQL, mantendo a segurança dos dados do usuário.

Envio de Dados

Após a validação, os dados do formulário são enviados para o servidor para processamento adicional. Isso geralmente é realizado por meio da submissão do formulário, onde os dados são encapsulados em uma solicitação HTTP e enviados para um endpoint específico no servidor. O servidor então processa os dados, executa as ações necessárias e retorna uma resposta ao cliente, completando o ciclo de interação entre o usuário e o servidor.

A tag **<form>** no HTML é uma ferramenta fundamental para a criação de formulários interativos em páginas da web. Esses formulários servem como mecanismos de coleta de informações dos usuários, permitindo uma interação direta e estruturada.

Ao utilizar a tag **<form>**, o desenvolvedor inicia o processo de definição e organização dos campos de entrada que comporão o formulário. Esses campos podem variar em natureza, desde caixas de texto simples para entrada de texto livre até menus suspensos para seleção de opções.

A inclusão de campos dentro do elemento **<form>** permite uma organização lógica e estruturada das informações a serem coletadas. Cada campo representa uma entidade específica de informação desejada, como nome, email, número de telefone, entre outros.

Além disso, o uso da tag **<form>** possibilita a especificação de ações a serem tomadas após o envio do formulário, por meio do atributo **action**. Este atributo define o destino para onde os dados coletados serão encaminhados para processamento posterior.

Por meio do emprego adequado da tag **<form>**, os desenvolvedores podem criar interfaces de usuário intuitivas e eficazes, facilitando a interação dos usuários com as páginas da web.

```
index.html > html > body > form > input
1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="UTF-8" />
5      <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6      <title>Elementos de Formulário HTML</title>
7      <style>
8        * {
9          margin: 3px;
10         padding: 3px;
11     }
12   </style>
13 </head>
14 <body>
15   <h1>Elementos de um formulário</h1>
16   <form>
17     <input type="text">
18   </form>
19 </body>
20 </html>
21
```

Elementos de um formulário



- **Adicionando Campos:** Dentro do formulário, você pode adicionar diferentes tipos de campos usando outras tags como `<input>`, `<textarea>`, `<select>`, etc. Cada campo é como uma pergunta que você está fazendo aos usuários.
- **Configurando a Ação:** Depois de adicionar todos os campos necessários, você precisa dizer ao navegador o que fazer com as informações quando o formulário for enviado.
- **Botão de Envio:** Por fim, você pode adicionar um botão de envio usando `<button>` ou `<input type="submit">`. Este botão é como o "Enviar" ou "Confirmar" em seu formulário. Quando clicado, ele envia todas as informações para o destino que você especificou.

Em HTML, a tag `<label>` é usada para criar rótulos para elementos de formulário, como campos de entrada (`<input>`), caixas de seleção (`<select>`), botões de rádio (`<input type="radio">`), etc.

A estrutura básica de uma tag de label é assim:

```
<label for="id_do_elemento">Texto do Rótulo:</label>
```

- **<label>:** Esta é a tag de abertura para criar um rótulo.
- **for="id_do_elemento":** Este atributo **for** é usado para associar o rótulo a um elemento específico no formulário. O valor desse atributo deve ser o mesmo que o atributo **id** do elemento de formulário ao qual você deseja associar o rótulo.
- **"Texto do Rótulo":** Este é o texto que aparecerá como rótulo no navegador. Você pode personalizá-lo conforme necessário.

```
index.html > html > body > form > input
1  <!DOCTYPE html>
2  <html lang="en">
3  | <head>
4  | | <meta charset="UTF-8" />
5  | | <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6  | | <title>Elementos de Formulário HTML</title>
7  | | <style>
8  | | | *
9  | | | | margin: 3px;
10 | | | | padding: 3px;
11 | | | }
12 | | </style>
13 | </head>
14 | <body>
15 | | <h1>Elementos de um formulário</h1>
16 | | <form>
17 | | | <label for="text">Texto:</label>
18 | | | <input type="text" id="text" />
19 |
20 | | | <br />
21 | | | <label for="">Senha:</label>
22 | | | <input type="password" />
23 | | </form>
24 | </body>
25 </html>
```

Elementos de um formulário

Texto:
Senha:



Aqui está uma explicação dos tipos de input HTML comumente usados em formulários JavaScript:

Input de Email (<input type="email">)

- Usado para solicitar que os usuários insiram um endereço de e-mail válido.
- Ao clicar neste tipo de botão, um campo de entrada é exibido onde os usuários podem digitar seu endereço de e-mail.
- Útil em formulários onde é necessário coletar informações de contato dos usuários, como ao se inscrever em newsletters, criar uma conta em um site, entre outros.

Input de Texto (<input type="text">):

- Este tipo de input é usado para coletar dados de texto do usuário, como nomes, endereços, mensagens, etc.
- Em JavaScript, você pode acessar e manipular o valor inserido neste input usando o objeto **value**.

Input de Senha (<input type="password">):

- Semelhante ao input de texto, mas os caracteres inseridos são ocultados, exibindo apenas asteriscos ou pontos.
- Útil para coletar senhas ou outras informações confidenciais.
- Para acessar seu valor em JavaScript, também use o objeto **value**.

Input de Número (<input type="number">):

- Este input permite que os usuários insiram apenas números.
- Além disso, ele pode incluir controles para aumentar ou diminuir o valor (como as setas para cima e para baixo).
- Em JavaScript, você pode acessar o valor usando **value** e converter para número conforme necessário.

Input de Checkbox (<input type="checkbox">):

- Usado para permitir que os usuários selecionem uma ou mais opções em um conjunto de opções.
- Cada checkbox tem um valor associado que pode ser acessado em JavaScript.
- Se o checkbox estiver marcado, seu valor é considerado verdadeiro; caso contrário, é falso.

Input de Radio (<input type="radio">):

- Semelhante ao checkbox, mas os usuários podem selecionar apenas uma opção de um conjunto de opções.
- Todos os radios no mesmo grupo devem ter o mesmo atributo **name** para que apenas um possa ser selecionado.
- A seleção pode ser acessada em JavaScript verificando qual radio está marcado.

Input de Arquivo (<input type="file">):

- Usado para permitir que os usuários selecionem um ou mais arquivos de seu dispositivo local para enviar para o servidor web.
- Ao clicar neste tipo de botão, uma janela de seleção de arquivo é aberta, permitindo que o usuário escolha o arquivo desejado.
- Útil em formulários onde os usuários precisam enviar documentos, imagens ou qualquer outro tipo de arquivo.

Input de Data (<input type="date">):

- Usado para permitir que os usuários selecionem uma data usando um calendário ou outro método específico do navegador.
- Ao clicar, um widget de calendário é exibido, permitindo que o usuário selecione uma data específica.
- Útil em formulários onde é necessário solicitar datas, como ao preencher informações de reserva, agendamento de compromissos, entre outros.

```
<form>
  <label for="text">Texto:</label>
  <input type="text" id="text" />

  <br />
  <label for="senha">Senha:</label>
  <input type="password" id="senha" />

  <br />
  <label for="checkbox">Checkbox</label>
  <input type="checkbox" id="checkbox" />

  <br />
  <label for="radio1">Radio1:</label>
  <input type="radio" id="radio1" name="radioGrupo" />

  <label for="radio2">Radio2:</label>
  <input type="radio" id="radio2" name="radioGrupo" />

  <br />
  <label for="file">File-Arquivo:</label>
  <input type="file" id="file" />

  <br />
  <label for="email">Email:</label>
  <input type="email" id="email" />
</form>
```

Elementos de um formulário

Texto:

Senha:

Checkbox

Radio1: Radio2:

File-Arquivo: Nenhum arquivo escolhido

Email:

```

  <label for="number">Número:</label>
  <input type="number" id="number" />

  <br />
  <label for="data">Data:</label>
  <input type="date" id="data" />
```

Número

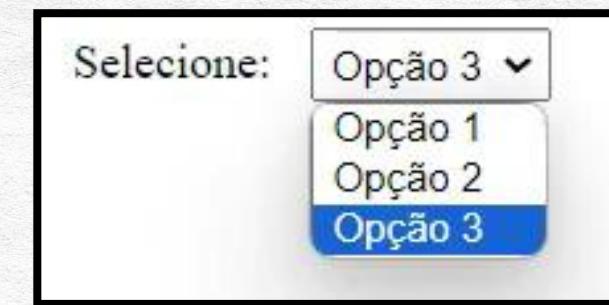
Data:

Elemento <select>:

- Usado para criar uma caixa de seleção dropdown em um formulário HTML.
- Permite que os usuários escolham uma ou mais opções de uma lista predefinida.
- Ao clicar no elemento **<select>**, uma lista de opções é exibida, e os usuários podem selecionar uma delas.
- Útil em formulários onde os usuários precisam selecionar uma opção de uma lista de escolhas, como escolher seu país de residência, estado civil, preferências de assinatura, entre outros.
- Pode ser combinado com o elemento **<option>** para definir as opções disponíveis na lista.
- No JavaScript, você pode acessar e manipular o valor selecionado do **<select>** para processar o formulário ou realizar outras ações com base na escolha do usuário.

Assim como outros elementos de entrada, o **<select>** oferece uma maneira conveniente de interagir com os usuários e coletar informações em formulários HTML.

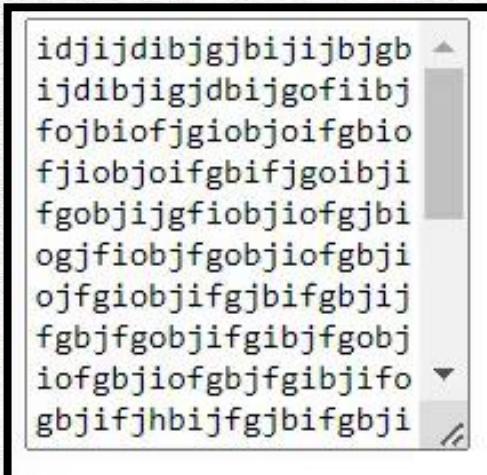
```
<label for="seleccione">Selecione:</label>
<select name="GrupoSelect" id="seleccione">
  <option value="opcao1">Opção 1</option>
  <option value="opcao2">Opção 2</option>
  <option value="opcao3">Opção 3</option>
</select>
```



Elemento <textarea>:

- Usado para criar uma área de texto multilinha em um formulário HTML.
- Permite que os usuários insiram texto longo, como comentários, mensagens, ou qualquer outro conteúdo que necessite de mais espaço do que um único campo de entrada.
- Ao clicar no elemento **<textarea>**, uma caixa de texto expandida é exibida, onde os usuários podem digitar seu texto.
- Útil em formulários onde é necessário coletar informações mais extensas dos usuários, como comentários em um blog, descrições de produtos em um formulário de pedido, entre outros.
- Pode ser definido com atributos como **rows** e **cols** para especificar o número de linhas e colunas visíveis na área de texto.
- No JavaScript, você pode acessar e manipular o conteúdo do **<textarea>** para processar o formulário ou realizar outras ações com base no texto inserido pelo usuário.

O **<textarea>** oferece uma maneira eficiente de coletar textos longos dos usuários em formulários HTML, proporcionando uma experiência de entrada de texto mais flexível do que um campo de entrada único.



```
<label for="textarea">Area de Texto:</label>
<textarea name="" id="textarea" cols="20" rows="10"></textarea>
```

Elementos <button> e <input> com tipo "submit":

- Ambos são usados para enviar um formulário para processamento quando clicados.
- O **<button>** com **type="submit"** e o **<input>** com **type="submit"** têm o mesmo comportamento padrão de submeter um formulário quando clicados.
- Ao clicar em um botão ou em um campo de entrada do tipo "submit", os dados do formulário são enviados ao servidor web para processamento.
- Útil em formulários onde os usuários precisam enviar informações para serem processadas, como formulários de login, registros, pesquisas, entre outros.
- No JavaScript, você pode associar funções aos eventos de clique desses elementos para realizar ações adicionais antes de enviar o formulário.

Diferenças entre <button> e <input> com tipo "submit":

- **<button>** permite mais personalização, como a inclusão de texto, imagens ou outros elementos HTML dentro do próprio botão.
- **<input>** com tipo "submit" é mais simples e direto, sendo apenas um botão de submissão.
- Em termos de acessibilidade, **<button>** é geralmente preferido, pois oferece mais flexibilidade para usuários de tecnologias assistivas, como leitores de tela.
- Em termos de estilo e design, **<button>** pode ser mais facilmente estilizado com CSS para se adequar ao design do site.

```
<label for="submit">Enviar</label>
<input type="submit" id="submit" />

<button type="submit">Enviar</button>
```



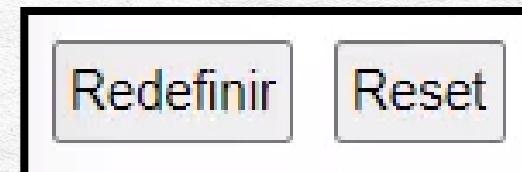
Elementos <button> e <input> com tipo "reset":

- Ambos são usados para redefinir os campos de um formulário para seus valores padrão quando clicados.
- O **<button>** com **type="reset"** e o **<input>** com **type="reset"** têm o mesmo comportamento padrão de redefinir os campos de um formulário para seus valores padrão quando clicados.
- Ao clicar em um botão ou em um campo de entrada do tipo "reset", todos os campos do formulário são limpos e retornam aos valores iniciais.
- Útil em formulários onde os usuários desejam limpar os campos preenchidos e recomeçar do zero.
- No JavaScript, você pode associar funções aos eventos de clique desses elementos para realizar ações adicionais antes de redefinir o formulário.

Diferenças entre <button> e <input> com tipo "reset":

- As diferenças entre **<button>** e **<input>** com tipo "reset" são semelhantes às diferenças entre eles com tipo "submit".
- **<button>** oferece mais flexibilidade e personalização em comparação com **<input>**.

```
<input type="reset" />  
<button type="reset">Reset</button>
```



Nesta aula, vamos implementar a estrutura de código ao lado dentro de um arquivo chamado index.html. Este código será responsável por criar um formulário simples, no qual vamos utilizar para adicionar 4 tipos de eventos de JavaScript: submit, focus, change e blur.

```
1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="UTF-8" />
5      <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6      <title>Formulário com Eventos em Javascript</title>
7      <style>
8        label {
9          display: block;
10         margin-bottom: 8px;
11       }
12     </style>
13   </head>
14   <body>
15     <h1>Formulário com Eventos em Javascript</h1>
16
17     <form id="meuFormulario">
18       <label for="campoTexto">Campo de Texto:</label>
19       <input type="text" id="campoTexto" />
20
21       <input type="submit" value="Enviar Dados" />
22     </form>
23
24     <script src="script.js"></script>
25   </body>
26 </html>
```

Formulário com Eventos em Javascript

Campo de Texto:

 Enviar Dados

O evento DOMContentLoaded é um evento JavaScript que é disparado quando todo o HTML foi completamente carregado e analisado, sem esperar pelo CSS, imagens ou outros recursos externos serem carregados. Isso significa que o DOM (Document Object Model) da página está pronto para ser manipulado via JavaScript.

Esse evento é útil quando você deseja executar algum código JavaScript assim que a estrutura básica da página estiver disponível, mas antes de todos os recursos adicionais, como imagens e folhas de estilo, serem completamente carregados. Isso permite que você inicie ações ou manipulações no DOM mais cedo, tornando a experiência do usuário mais responsiva.

```
document.addEventListener('DOMContentLoaded', function() {
    // Código a ser executado quando o DOM estiver completamente carregado
    console.log('O DOM está pronto!');
});
```

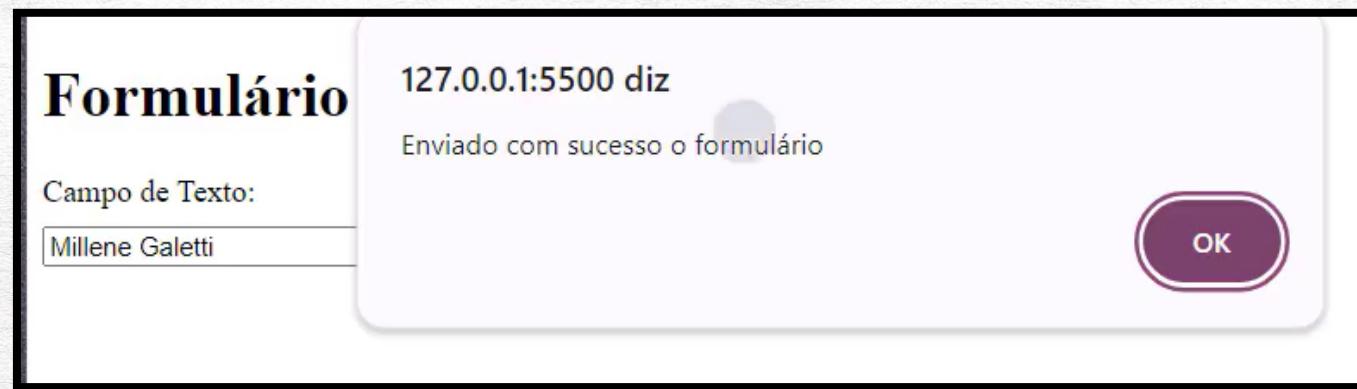
O evento "submit" é um evento que ocorre quando um formulário HTML é enviado. Quando um usuário clica no botão de envio dentro de um formulário, o navegador dispara o evento "submit". Este evento é útil para validar os dados do formulário antes de enviá-los para o servidor.

```
let formulario = document.getElementById("meuFormulario");
formulario.addEventListener("submit", function () {
    alert("Dados enviados");
});
```



- **let formulario = document.getElementById("meuFormulario");**: Este trecho de código obtém o elemento do formulário HTML com o ID "meuFormulario" e armazena-o na variável "formulario".
- **formulario.addEventListener("submit", function () { alert("Dados enviados"); });**: Aqui, você está adicionando um ouvinte de eventos ao formulário. Quando o evento "submit" ocorre no formulário, a função anônima (função sem nome) é executada. Neste caso, a função exibe um alerta com a mensagem "Dados enviados".

Portanto, sempre que o formulário for enviado (por exemplo, quando o usuário clica no botão de envio), o alerta "Dados enviados" será exibido. Isso pode ser útil para fornecer feedback ao usuário de que os dados foram enviados com sucesso, ou para executar outras ações relacionadas ao envio do formulário, como validação de dados.



- **let formulario = document.getElementById("meuFormulario");**: Este trecho de código obtém o elemento do formulário HTML com o ID "meuFormulario" e armazena-o na variável "formulario".
- **formulario.addEventListener("submit", function () { alert("Dados enviados"); });**: Aqui, você está adicionando um ouvinte de eventos ao formulário. Quando o evento "submit" ocorre no formulário, a função anônima (função sem nome) é executada. Neste caso, a função exibe um alerta com a mensagem "Dados enviados".

Portanto, sempre que o formulário for enviado (por exemplo, quando o usuário clica no botão de envio), o alerta "Dados enviados" será exibido. Isso pode ser útil para fornecer feedback ao usuário de que os dados foram enviados com sucesso, ou para executar outras ações relacionadas ao envio do formulário, como validação de dados.

O próximo trecho de código JavaScript está lidando com um evento de mudança ("change") em um campo de texto HTML. Aqui está a explicação linha por linha:

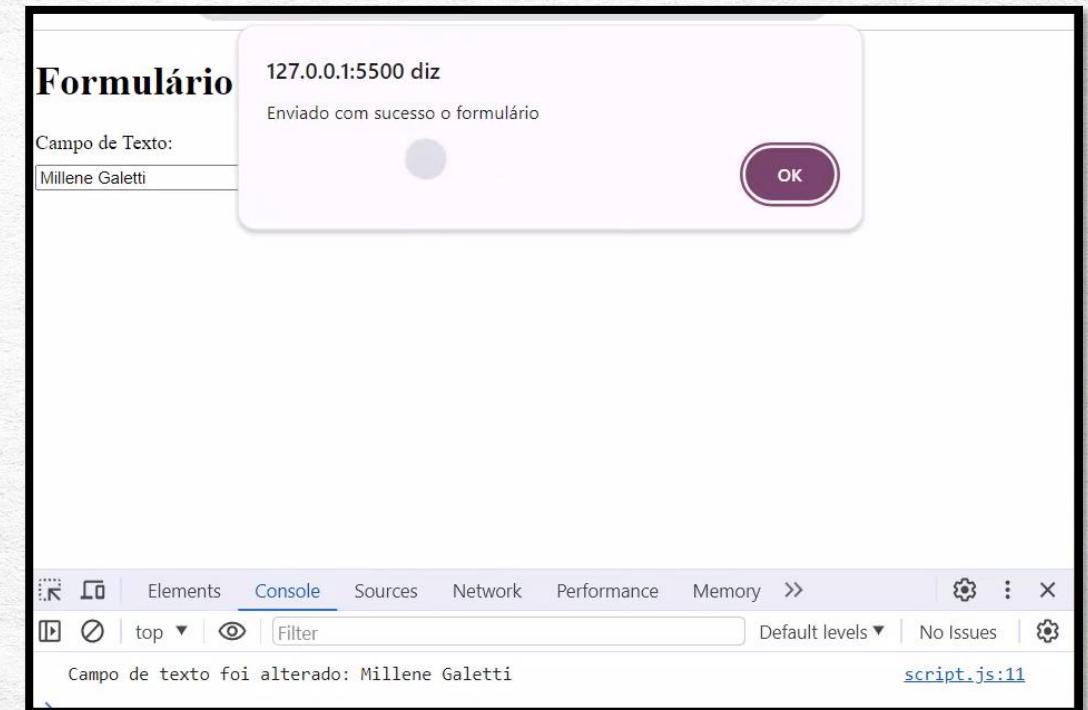
```
let campoTexto = document.getElementById("campoTexto");
```

- **let campoTexto**: Aqui, você está declarando uma variável chamada "campoTexto" usando o comando "let". Essa variável será usada para armazenar uma referência ao elemento HTML do campo de texto que possui o ID "campoTexto".
- **document.getElementById("campoTexto")**: Este trecho de código está procurando no documento HTML um elemento com o ID "campoTexto". Presumivelmente, esse elemento é um campo de texto HTML.

```
campoTexto.addEventListener("change", function () {  
    console.log("Campo de Texto foi alterado: " + campoTexto.value)  
});
```

- **campoTexto.addEventListener("change", function () { ... })**: Aqui, você está adicionando um ouvinte de eventos ao campo de texto. Especificamente, você está ouvindo pelo evento "change", que ocorre quando o conteúdo do campo de texto é alterado pelo usuário.
- **function () { ... }**: Este é um exemplo de uma função anônima sendo passada como argumento para o método "addEventListener". Essa função será chamada sempre que o evento "change" ocorrer no campo de texto.
- **console.log("Campo de Texto foi alterado: " + campoTexto.value)**: Dentro da função anônima, você está registrando uma mensagem no console do navegador. A mensagem inclui o texto "Campo de Texto foi alterado:" seguido pelo valor atual do campo de texto, acessado através da propriedade "value" do elemento "campoTexto".

Portanto, sempre que o usuário alterar o conteúdo do campo de texto identificado pelo ID "campoTexto", uma mensagem será registrada no console do navegador, indicando que o campo de texto foi alterado e exibindo o novo valor do campo. Isso pode ser útil para rastrear as interações do usuário com o campo de texto.



```
campoTexto.addEventListener("focus", function () {  
    console.log("Campo de Texto Focado!");  
});
```

Este trecho de código JavaScript está lidando com o evento de foco ("focus") em um campo de texto HTML.

- **campoTexto.addEventListener("focus", function () { ... })**: Aqui, você está adicionando um ouvinte de eventos ao campo de texto. Especificamente, você está ouvindo pelo evento "focus", que ocorre quando o campo de texto recebe o foco, ou seja, quando ele é selecionado pelo usuário ou programaticamente.
- **function () { ... }**: Esta é uma função anônima que será chamada sempre que o evento "focus" ocorrer no campo de texto.
- **console.log("Campo de Texto Focado!")**: Dentro da função anônima, você está registrando uma mensagem no console do navegador. Esta mensagem indica que o campo de texto foi focado.

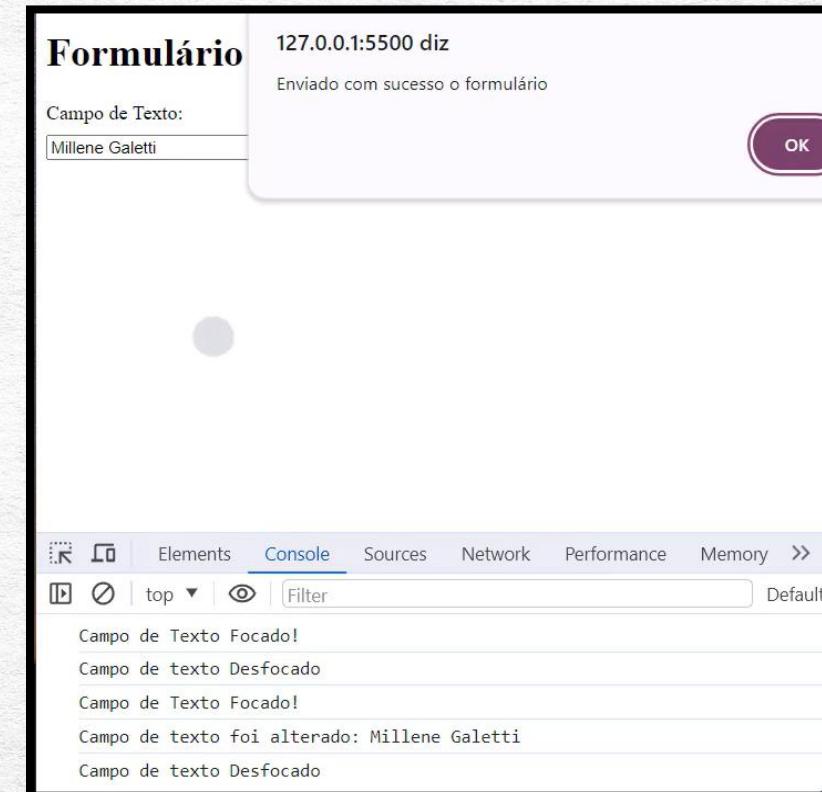
Portanto, sempre que o usuário ou algum script der foco ao campo de texto, uma mensagem será registrada no console do navegador indicando que o campo de texto foi focado. Isso pode ser útil para rastrear interações do usuário com o campo de texto ou para realizar ações específicas quando o campo recebe o foco.

Os eventos em JavaScript podem ocorrer em uma ordem que não necessariamente corresponde à ordem em que estão escritos no código. Isso ocorre porque JavaScript é uma linguagem assíncrona, o que significa que várias operações podem estar ocorrendo simultaneamente, e o navegador pode lidar com eventos de forma não sequencial, dependendo de vários fatores, como a interação do usuário, o carregamento de recursos externos e outras operações em segundo plano.

```
campoTexto.addEventListener("blur", function () {  
    console.log("Campo de Texto Desfocado!");  
});
```

- **campoTexto.addEventListener("blur", function () { ... }):** Aqui, você está adicionando um ouvinte de eventos ao campo de texto. Especificamente, você está ouvindo pelo evento "blur", que ocorre quando o campo de texto perde o foco, ou seja, quando ele deixa de ser selecionado pelo usuário ou programaticamente.
- **function () { ... }:** Esta é uma função anônima que será chamada sempre que o evento "blur" ocorrer no campo de texto.
- **console.log("Campo de Texto Desfocado!"):** Dentro da função anônima, você está registrando uma mensagem no console do navegador. Esta mensagem indica que o campo de texto perdeu o foco.

Sempre que o usuário ou algum script remover o foco do campo de texto, uma mensagem será registrada no console do navegador indicando que o campo de texto foi desfocado. Isso pode ser útil para rastrear interações do usuário com o campo de texto ou para realizar ações específicas quando o campo perde o foco.



Para entendermos sobre manipulação de formulários, vamos estruturar um arquivo chamado "index.html" com a estrutura abaixo:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Manipulação de eventos</title>
    <style>
      label {
        display: block;
        margin-bottom: 8px;
      }
    </style>
  </head>
  <body>
    <h1>Manipulação do Formulário - Eventos</h1>

    <form id="meuFormulario">
      <label for="campoTexto">Campo de Texto:</label>
      <input type="text" id="campoTexto" />

      <button type="button" onclick="mostrarValorCampo()">Enviar Dados</button>
    </form>

    <script src="script.js"></script>
  </body>
</html>
```

E agora vamos implementar dentro de um arquivo script.js a funcionalidade para o nosso evento HTML onclick, chamado de "mostrarValorCampo()".

Manipulação do Formulário - Eventos

Campo de Texto:

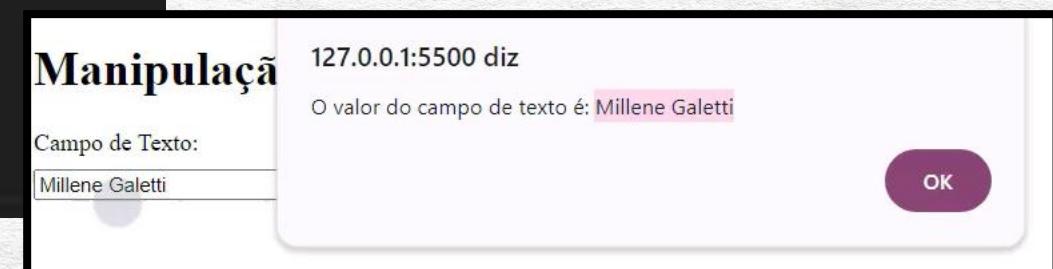
 Enviar Dados

Os formulários HTML são uma parte fundamental das interações do usuário em páginas da web. Eles permitem aos usuários inserir dados e enviá-los para o servidor. Uma maneira de tornar esses formulários mais dinâmicos e interativos é através da manipulação de eventos diretamente no HTML, como usando o atributo **onclick** em um botão.

O atributo **onclick** permite associar uma função JavaScript a um evento de clique em um elemento HTML, como um botão. Isso significa que quando o usuário clica no botão, a função especificada é executada, permitindo-nos realizar ações específicas em resposta ao clique.

Usar eventos diretos no HTML, como o **onclick**, é uma maneira simples e eficaz de adicionar interatividade aos formulários, permitindo que respondamos às ações do usuário de forma rápida e dinâmica.

```
function mostrarValorCampo() {  
    // Acessar o valor do campo de texto = propriedade .value  
    let valorCampo = document.getElementById("campoTexto").value;  
  
    //Exibir o valor em um alerta  
    alert("O valor do campo de texto é: " + valorCampo);  
}
```



Validação de Formulários

A validação de formulários é essencial no desenvolvimento web, independentemente do nível de experiência. Ela garante que os dados inseridos pelos usuários estejam corretos, completos e seguros. Imagine um formulário de cadastro onde um usuário digita seu e-mail, mas comete um erro de digitação, esquece de preencher um campo obrigatório ou insere informações em um formato inválido. Sem validação, esses erros podem passar despercebidos e comprometer a funcionalidade do sistema.

Ao criar formulários sem validação, os desenvolvedores correm o risco de receber dados incorretos ou incompletos, o que pode levar a consequências negativas, como registros duplicados, dificuldades na filtragem e processamento de dados, ou até mesmo vulnerabilidades de segurança.

Portanto, a validação de formulários é crucial para garantir a integridade dos dados e proporcionar uma experiência de usuário consistente e confiável. Ela ajuda a evitar erros comuns de entrada de dados e contribui para a eficiência e segurança das aplicações web. Além disso, ao implementar a validação de forma adequada, os desenvolvedores podem economizar tempo e recursos, evitando retrabalhos causados por problemas de dados incorretos.

Então por que é importante aprender sobre validação de formulários? A resposta é simples: a validação de formulários ajuda a garantir a precisão e a consistência dos dados submetidos pelos usuários.

Este código HTML representa um formulário simples com um campo de texto e um botão de envio de dados.

- **<form id="meuFormulario">**: Aqui temos a tag **<form>**, que é usada para criar um formulário em HTML. O atributo **id** define um identificador único para o formulário, neste caso, "meuFormulario". Isso pode ser útil para referenciar o formulário em scripts JavaScript ou em estilos CSS.
- **<label for="campoTexto">Campo de Texto:</label>**: A tag **<label>** é usada para rotular um campo de formulário, neste caso, o campo de texto. O atributo **for** estabelece uma associação entre o rótulo e o campo de entrada correspondente, usando o valor do atributo **id** do campo de entrada. Assim, quando o usuário clica no rótulo, o foco é direcionado para o campo de texto associado.
- **<input type="text" id="campoTexto" />**: Aqui temos a tag **<input>**, que cria um campo de entrada no formulário. O atributo **type** especifica o tipo de entrada, neste caso, "text" para um campo de texto. O atributo **id** define um identificador único para o campo de texto, permitindo referenciá-lo de outras partes do código. O usuário pode inserir texto neste campo.
- **<input type="submit" value="Enviar Dados" />**: Esta é outra tag **<input>**, mas desta vez do tipo "submit", que cria um botão de envio no formulário. Quando o usuário clica neste botão, os dados inseridos no formulário são enviados para o servidor para processamento. O atributo **value** define o texto exibido no botão, neste caso, "Enviar Dados".

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Validação de Formulário</title>
    <style>
        label {
            display: block;
            margin-bottom: 8px;
        }
    </style>
</head>
<body>
    <h1>Validação de Formulário - parte 1</h1>

    <form id="meuFormulario" onsubmit="return validarFormulario()">
        <label for="campoTexto">Campo de Texto:</label>
        <input type="text" id="campoTexto" />

        <input type="submit" value="Enviar Dados" />
    </form>

    <script src="script.js"></script>
</body>
</html>
```

- **onsubmit="return validarFormulario()":** Este é um atributo do formulário chamado **onsubmit**. Ele é um evento que ocorre quando o formulário é enviado. No caso, quando o formulário é submetido (quando o usuário clica no botão de envio), o navegador executa o código JavaScript especificado no valor desse atributo.
 - **return validarFormulario():** Aqui, **validarFormulario()** é uma função JavaScript que é chamada quando o formulário é enviado. O **return** antes da chamada da função indica que estamos esperando um retorno booleano (verdadeiro ou falso) da função **validarFormulario()**. Se essa função retornar **true**, o formulário será enviado normalmente; se retornar **false**, o envio do formulário será cancelado.

```
<form id="meuFormulario" onsubmit="return validarFormulario()">
```

Portanto, o código HTML em questão está associando a função **validarFormulario()** ao evento de submissão do formulário. Isso permite que o desenvolvedor valide os dados do formulário antes de serem enviados para o servidor, garantindo assim a integridade e a precisão dos dados submetidos pelos usuários.

```
function validarFormulario() {  
    //Obter o valor do campo de texto  
    const campoTexto = document.getElementById("campoTexto");  
    let valorCampo = campoTexto.value;  
  
    //Verificar se o campo de texto está vazio ou não  
    if (valorCampo === "") {  
        alert("Por favor, preencha o campo de texto!");  
        return false; // impede a submissão do formulário  
    } else {  
        return true; // permite a submissão do formulário  
    }  
}
```

function validarFormulario() { ... }: Esta é a declaração de uma função em JavaScript chamada **validarFormulario()**. Essa função será chamada quando o formulário for submetido.

const campoTexto = document.getElementById("campoTexto");: Esta linha obtém o elemento HTML com o **id** "campoTexto" usando o método **document.getElementById()**. Esse elemento é o campo de texto do formulário.

let valorCampo = campoTexto.value;: Aqui, o valor atual do campo de texto é obtido usando a propriedade **value**. Esse valor será usado para verificar se o campo está vazio ou não.

if (valorCampo === "") { ... } else { ... }: Esta estrutura condicional verifica se o campo de texto está vazio. Se estiver vazio (**valorCampo === ""**), um alerta é exibido ao usuário solicitando que preencha o campo de texto. Em seguida, a função retorna **false**, o que impede a submissão do formulário. Se o campo não estiver vazio, a função retorna **true**, permitindo a submissão do formulário.

- Esta função **validarFormulario()** é responsável por verificar se o campo de texto do formulário está preenchido ou não. Se estiver vazio, o usuário é alertado e a submissão do formulário é impedida; caso contrário, a submissão é permitida. Essa validação ajuda a garantir que os dados inseridos pelo usuário sejam válidos antes de serem enviados para o servidor.

A partir desta aula, vamos implementar os seguintes exemplos de validações de formulário. Para isso, iremos construir os elementos do nosso formulário dentro do arquivo index.html, assim como a lógica e as funcionalidades de validação dentro de um arquivo script.js que será importado para o arquivo HTML.

Formulário de Registro

Nome: Este campo é obrigatório.

Email: Este campo é obrigatório.

Senha: Este campo é obrigatório.

Confirmar Senha: Este campo é obrigatório.

Formulário de Registro

Nome:

Email:

Senha:

Confirmar Senha: As senhas não coincidem.

Vamos dividir a nossa implementação de código dentro do arquivo HTML em partes para entender melhor o que está acontecendo.

1. Estrutura HTML

```
<h1>Validação de Campos em Javascript</h1>

<form id="meuFormulario">
  <div class="form-group">
    <label for="nome">Nome:</label>
    <input type="text" id="nome" class="obrigatorio" />
    <span class="error-message"></span>
  </div>
  <!-- Repetido para email, senha e confirmação de senha -->

  <button type="submit">Enviar</button>
</form>
```

- Temos um formulário HTML (**<form>**) com um título e campos para nome, email, senha e confirmação de senha.
- Cada campo (**<input>**) tem um **id** para identificação única e uma classe **obrigatorio**.
- Há também uma **span** vazia para exibir mensagens de erro.

2. Estilização CSS

```
.form-group {  
    margin-bottom: 20px; /* Espaçamento entre grupos de campos */  
}  
  
.error-message {  
    color: red;          /* Cor do texto de erro */  
    font-size: 12px;      /* Tamanho da fonte do texto de erro */  
    display: none;        /* Inicialmente, as mensagens de erro estarão ocultas */  
}
```

- Define estilos para os grupos de campos e mensagens de erro.
- As mensagens de erro são inicialmente ocultas (**display: none**) para que só apareçam quando houver um erro de validação.

3. O que o formulário está fazendo?

Este formulário, quando submetido, irá tentar validar se os campos estão preenchidos. Isso é feito através de JavaScript, embora não esteja presente no código fornecido.

Vamos adicionar código JavaScript para lidar com a validação na próxima aula.

- **Nome:** Deve ser preenchido.
- **Email:** Deve ser um endereço de email válido.
- **Senha:** Deve ser preenchida.
- **Confirmar Senha:** Deve ser igual à senha.

Se algum desses critérios não for atendido, uma mensagem de erro será exibida abaixo do campo correspondente.

A estilização fornecida ajuda a tornar a interface do formulário mais agradável e intuitiva, fornecendo feedback visual sobre os erros de validação.

```
<body>
  <h1>Validação de Campos em Javascript</h1>

  <form id="meuFormulario">
    <div class="form-group">
      <label for="nome">Nome:</label>
      <input type="text" id="nome" class="obrigatorio" />
      <span class="error-message"></span>
    </div>
    <div class="form-group">
      <label for="email">Email:</label>
      <input type="email" id="email" class="obrigatorio" />
      <span class="error-message"></span>
    </div>
    <div class="form-group">
      <label for="senha">Senha:</label>
      <input type="password" id="senha" class="obrigatorio" />
      <span class="error-message"></span>
    </div>
    <div class="form-group">
      <label for="confirmarSenha">Confirmar Senha:</label>
      <input type="password" id="confirmarSenha" class="obrigatorio" />
      <span class="error-message"></span>
    </div>

    <button type="submit">Enviar</button>
  </form>
  <script src=".//script.js"></script>
</body>
```



Adicionando a classe "obrigatorio"

Quando adicionamos a classe "obrigatorio" aos elementos <input>, estamos sinalizando que esses campos são obrigatórios para o envio do formulário.

```
<input type="text" id="nome" class="obrigatorio" />
```

Agora vamos começar a implementar o nosso Javascript para validar os campos.

```
document.addEventListener("DOMContentLoaded", function () {
  const formulario = document.getElementById("meuFormulario");

  formulario.addEventListener("submit", function (event) {
    validarCamposObrigatorios();
    event.preventDefault();

    function validarCamposObrigatorios() {
      let camposObrigatorios = document.querySelectorAll(".obrigatorio");
      console.log(camposObrigatorios);
    };
  });
});
```

```
document.addEventListener("DOMContentLoaded", function () {
  ...
});
```

- **DOMContentLoaded** é um evento que é acionado quando o HTML foi completamente carregado e analisado, sem esperar pelo CSS, imagens, e outros recursos externos serem carregados. Isso significa que é seguro acessar os elementos da página neste ponto.
- O código dentro dessa função será executado assim que o evento **DOMContentLoaded** for acionado.

```
const formulario = document.getElementById("meuFormulario");
```

- Aqui, estamos selecionando o elemento do formulário com o id "meuFormulario" e armazenando-o na variável **formulario**.
- Isso nos permite acessar e manipular esse formulário facilmente no código JavaScript.



formulario.addEventListener("submit", function (event) { ... });

- Adicionamos um event listener para o evento de "submit" do formulário.
- Quando o formulário é submetido (quando o botão de envio é clicado), a função passada como segundo argumento será executada.

validarCamposObrigatorios();

- Dentro do event listener de "submit", chamamos a função **validarCamposObrigatorios()**.
- Esta função é responsável por validar os campos obrigatórios do formulário.

event.preventDefault();

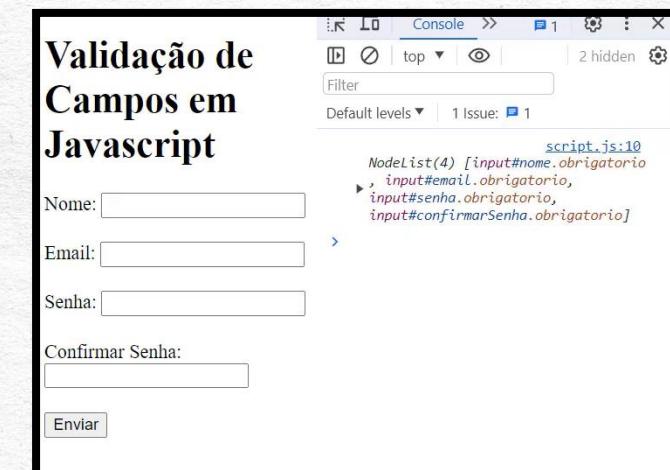
- Logo após a chamada da função **validarCamposObrigatorios()**, chamamos **event.preventDefault()** para evitar que o formulário seja enviado (recarregando a página) antes de realizarmos a validação e tomar as ações necessárias.

function validarCamposObrigatorios() { ... }

- Esta é a função que definimos para validar os campos obrigatórios.
- Dentro dela, selecionamos todos os elementos com a classe "obrigatorio" usando **document.querySelectorAll(".obrigatorio")**.
- Esses elementos são armazenados na variável **camposObrigatorios**.
- No exemplo dado, apenas imprimimos esses campos no console para fins de depuração.

Resumindo

- Espera que todo o HTML seja carregado.
- Seleciona o formulário pelo ID.
- Adiciona um event listener para o evento "submit" do formulário.
- Dentro do event listener, chama uma função para validar os campos obrigatórios.
- Evita o envio do formulário antes da validação ser realizada.
- A função de validação identifica os campos obrigatórios e os armazena em uma variável.



Agora, vamos implementar uma validação para garantir que todos os campos obrigatórios do formulário sejam preenchidos antes de permitir o envio. Vamos dar uma olhada no código:

```
document.addEventListener("DOMContentLoaded", function () {
  const formulario = document.getElementById("meuFormulario");

  formulario.addEventListener("submit", function (event) {
    if (!validarCamposObrigatorios()) {
      event.preventDefault(); // impede a submissão do formulário se houver erros!
    }
  });

  function validarCamposObrigatorios() {
    let camposObrigatorios = document.querySelectorAll(".obrigatorio");
    // console.log(camposObrigatorios);
    let camposValidos = true;

    for (let i = 0; camposObrigatorios.length > i; i++) {
      let campo = camposObrigatorios[i];

      if (campo.value === "" || campo.value === null) {
        console.log("Este campo é obrigatório");
        camposValidos = false;
      }
    }
    return camposValidos;
  }
});
```

- **if (!validarCamposObrigatorios()) { ... }:** Aqui, estamos verificando se a função **validarCamposObrigatorios()** retorna **false**. Se isso acontecer, significa que os campos obrigatórios do formulário não foram preenchidos corretamente. Nesse caso, estamos usando **event.preventDefault()** para evitar que o formulário seja enviado.
- **function validarCamposObrigatorios() { ... }:** Esta é a função que valida se os campos obrigatórios do formulário estão preenchidos. Ela retorna **true** se todos os campos obrigatórios estiverem preenchidos e **false** se algum deles estiver vazio.
- **let camposObrigatorios = document.querySelectorAll(".obrigatorio");**: Aqui, estamos selecionando todos os elementos HTML que têm a classe CSS "obrigatorio" e armazenando-os em **camposObrigatorios**.

- **for (let i = 0; camposObrigatorios.length > i; i++) { ... }:** Este é um loop que itera sobre todos os campos obrigatórios encontrados no passo anterior.
- **let campo = camposObrigatorios[i];**: Dentro do loop, estamos armazenando o campo atual em **campo**, para facilitar a referência posterior.
- **if (campo.value === "" || campo.value === null) { ... }:** Aqui, estamos verificando se o valor do campo atual está vazio ("") ou **null**. Se estiver vazio, isso significa que o campo obrigatório não foi preenchido, então definimos **camposValidos** como **false**.
- **return camposValidos;**: Finalmente, retornamos o valor de **camposValidos** após o loop ter sido concluído. Isso indicará se todos os campos obrigatórios foram preenchidos corretamente ou não.



Validação de Campos Obrigatórios: Suponha que temos um formulário HTML com vários campos, alguns dos quais são obrigatórios para serem preenchidos pelo usuário antes de enviar o formulário. A função **validarCamposObrigatorios()** provavelmente percorreria esses campos e verificaria se estão preenchidos corretamente. Se algum campo obrigatório não estiver preenchido ou não estiver de acordo com as regras definidas, a função retornará **false**, indicando que a validação falhou.

Operador ! (Negação): O operador **!** é usado para negar o valor de uma expressão booleana. Quando você usa **!** antes de uma expressão, ele inverte o valor booleano dessa expressão. Por exemplo:

- Se **validarCamposObrigatorios()** retorna **true**, então **!validarCamposObrigatorios()** será **false**.
- Se **validarCamposObrigatorios()** retorna **false**, então **!validarCamposObrigatorios()** será **true**.

Condição de Execução do Bloco de Código: O código **if (!validarCamposObrigatorios()) { ... }** usa a negação para verificar se a função **validarCamposObrigatorios()** retorna **false**. Isso significa que o bloco de código dentro das chaves **{ }** será executado apenas se os campos obrigatórios não estiverem preenchidos corretamente, ou seja, se a validação falhar.

Então, basicamente, esse trecho de código está dizendo: "Se os campos obrigatórios não estiverem preenchidos corretamente, previna o comportamento padrão de submissão do formulário." Isso é feito através do **event.preventDefault()**, que impede o envio do formulário quando a validação falha.

```
if (!validarCamposObrigatorios()) {
    event.preventDefault(); // impede a submissão do formulário se houver erros!
}
```

Separar funções e fazer chamadas dentro de uma função principal é uma prática importante na programação por uma série de razões. Vamos analisar o código fornecido e discutir por que essa abordagem é benéfica.

No código fornecido, temos um event listener que escuta o evento de envio (submit) de um formulário. Quando o formulário é submetido, a função anônima associada é chamada. Dentro dessa função, há uma chamada para a função **validarCamposObrigatorios()**.

Aqui estão alguns motivos para separar funções e chamar apenas dentro de uma função "principal":

- **Legibilidade e organização do código:** Ao separar as funcionalidades em funções menores e autocontidas, o código se torna mais legível e fácil de entender. Isso facilita a manutenção e a colaboração com outros desenvolvedores.
- **Reutilização de código:** Se uma funcionalidade específica precisa ser executada em diferentes partes do programa, é mais eficiente e econômico ter essa funcionalidade encapsulada em uma função que pode ser chamada em vários lugares. No caso do código que estamos trabalhando em aula, se houver necessidade de validar campos obrigatórios em outros formulários, a função **validarCamposObrigatorios()** pode ser reutilizada.
- **Testabilidade:** Separar as funcionalidades em funções individuais facilita a realização de testes automatizados. Você pode escrever testes unitários para cada função separadamente, garantindo que cada parte do seu código funcione corretamente.
- **Modularidade e manutenção:** Quando as funcionalidades estão separadas em funções individuais, é mais fácil fazer alterações em partes específicas do código sem afetar o restante do programa. Isso promove uma base de código mais modular e facilita a manutenção a longo prazo.

A função **validarCamposObrigatorios()** é responsável por verificar se os campos obrigatórios estão preenchidos no formulário. Ao separar essa funcionalidade em uma função separada, você está seguindo os princípios de boas práticas de programação, tornando seu código mais legível, modular e fácil de manter.

```
1 > document.addEventListener("DOMContentLoaded", function () {
2   const formulario = document.getElementById("meuFormulario");
3
4 >   formulario.addEventListener("submit", function (event) { ...
5   });
6
7 >   function validarCamposObrigatorios() { ...
8   }
9
10 });

11 > 
```



```
function compararValores() {  
    const senha = document.getElementById("senha");  
    const confirmarSenha = document.getElementById("confirmarSenha");  
  
    if (senha.value !== confirmarSenha.value) {  
        exibirErro(confirmarSenha, "As senhas não coincidem.");  
        return false;  
    }  
    return true;  
}
```

Essa função JavaScript chamada **compararValores()** é responsável por comparar dois valores de entrada em um formulário HTML, geralmente relacionados a senhas. Vamos analisá-la passo a passo:

- **Atribuição de variáveis:** A função começa obtendo referências aos elementos do formulário HTML com os IDs "senha" e "confirmarSenha". Ela usa o método **getElementById** do objeto **document** para fazer isso. Isso significa que deve haver elementos no HTML com esses IDs.

Comparação de valores: Em seguida, a função compara os valores dos campos de entrada das senhas (**senha** e **confirmarSenha**). Se esses valores forem diferentes, a função exibe uma mensagem de erro utilizando outra função chamada **exibirErro()** e retorna **false**, indicando que a comparação falhou.

Retorno de resultado: Se os valores das senhas forem iguais, a função simplesmente retorna **true**, indicando que a comparação foi bem-sucedida.

Este trecho de código JavaScript está sendo usado em conjunto com a função **compararValores()** para validar um formulário antes de ser enviado. Vamos analisar o que está acontecendo aqui:

- **Chamada da função compararValores():** A linha **if (!compararValores()) {** chama a função **compararValores()** e verifica se o resultado retornado é **false**. O operador **!** nega o resultado da função, então **!compararValores()** significa "se a função **compararValores()** retornar **false**".
- **Bloqueio do evento padrão:** Se a função **compararValores()** retornar **false**, isso significa que a comparação de valores falhou, provavelmente porque as senhas não coincidem. Nesse caso, o código dentro do bloco **if** é executado. O método **preventDefault()** é chamado no objeto **event** para evitar que o comportamento padrão do evento ocorra.
 - Por exemplo, se este código estiver dentro de um manipulador de evento para o envio de um formulário (**submit**), **event.preventDefault()** impedirá que o formulário seja enviado caso as senhas não coincidam. Isso é útil para fornecer feedback ao usuário e evitar que dados incorretos sejam enviados.

```
document.addEventListener("DOMContentLoaded", function () {
    const formulario = document.getElementById("meuFormulario");

    formulario.addEventListener("submit", function (event) {

        if (!validarCamposObrigatorios()) {
            event.preventDefault(); // impede a submissão do formulário se houver erros!
        }

        if (!compararValores()) {
            event.preventDefault();
        }
    });
});
```



Nesta aula, vamos implementar a criação de uma lógica para as nossas mensagens de erro, que vão substituir os nossos **console.logs**, os quais estavam mostrando as mensagens de validação do nosso formulário.

```
function exibirErro(elemento, mensagem) {
  let mensagemErro = elemento.parentElement.querySelector(".error-message");
  mensagemErro.textContent = mensagem;
  mensagemErro.style.display = "inline-block"; // exibir a mensagem de erro
}
```

Essa função **exibirErro** recebe dois parâmetros: **elemento** e **mensagem**.

- **elemento**: Este parâmetro representa um elemento HTML do qual o erro será associado.
- **mensagem**: Este parâmetro é a mensagem de erro que será exibida ao usuário.

A função busca dentro do elemento pai do elemento fornecido (**elemento.parentElement**) por um elemento que tenha a classe **.error-message** utilizando o método **querySelector**. Essa classe provavelmente é usada para estilizar a mensagem de erro.

Em seguida, a função atualiza o conteúdo desse elemento **.error-message** com a mensagem de erro fornecida (**mensagemErro.textContent = mensagem**). Finalmente, define o estilo CSS **display** desse elemento como "**inline-block**", tornando-o visível na página (**mensagemErro.style.display = "inline-block"**).

Agora vamos substituir nossos consoles, com a chamada dessa nova função, e adicionar os argumentos necessários:

```
function validarCamposObrigatorios() {
  let camposObrigatorios = document.querySelectorAll(".obrigatorio");
  // console.log(camposObrigatorios);
  let camposValidos = true;

  for (let i = 0; camposObrigatorios.length > i; i++) {
    let campo = camposObrigatorios[i];

    if (campo.value === "" || campo.value === null) {
      exibirErro(campo, "Este campo é obrigatório");
      camposValidos = false;
    }
  }
  return camposValidos;
}
```

```
function compararValores() {
  const senha = document.getElementById("senha");
  const confirmarSenha = document.getElementById("confirmarSenha");

  if (senha.value !== confirmarSenha.value) {
    exibirErro(confirmarSenha, "As senhas não coincidem.");
    return false;
  }
  return true;
}
```

Vamos adicionar a última funcionalidade do nosso formulário, que será a capacidade de resetar a mensagem de erro quando o usuário preencher um campo obrigatório:

```
function resetarMensagensDeErro() {
  let mensagensErro = document.querySelectorAll(".error-message");
  //console.log(mensagensErro);

  for (let i = 0; i < mensagensErro.length; i++) {
    mensagensErro[i].textContent = "";
  }
}
```

Essa função **resetarMensagensDeErro** não recebe nenhum parâmetro. Ela é responsável por limpar todas as mensagens de erro que estão presentes na página.

- Primeiro, a função utiliza **document.querySelectorAll(".error-message")** para selecionar todos os elementos na página que possuem a classe **.error-message**. Isso retorna uma NodeList contendo todos esses elementos.
- Em seguida, a função itera sobre essa NodeList usando um loop **for**, indo de 0 até o comprimento da lista de elementos (**mensagensErro.length**). Para cada elemento na lista, o conteúdo de texto (**textContent**) é definido como uma string vazia "", o que efetivamente remove o texto de erro que estava sendo exibido.

Essa função é útil para resetar todas as mensagens de erro em um formulário ou em qualquer parte da página onde as mensagens de erro são exibidas, permitindo uma limpeza completa antes de uma nova validação ou interação com o usuário.

```
formulario.addEventListener("submit", function (event) {
    // Resetando mensagens erro
    resetarMensagensDeErro();

    if (!validarCamposObrigatorios()) {
        event.preventDefault(); // impede a submissão do formulário se houver erros!
    }

    if (!compararValores()) {
        event.preventDefault();
    }
});
```

Nesse trecho do código, a função **resetarMensagensDeErro()** é chamada dentro do evento de submissão de um formulário.

Quando o formulário é submetido (através do evento "submit"), esta função é invocada primeiro.

A razão para isso é garantir que todas as mensagens de erro anteriores sejam removidas antes de qualquer validação ocorrer. Isso é importante porque, ao submeter o formulário novamente, queremos garantir que o usuário não veja mensagens de erro antigas que não são mais relevantes para os campos que ele modificou.

Então, o fluxo do código é o seguinte:

- **Resetar Mensagens de Erro:** A função **resetarMensagensDeErro()** é chamada para limpar todas as mensagens de erro que podem estar presentes na página.
- **Validação dos Campos Obrigatórios:** Em seguida, o código executa uma função **validarCamposObrigatorios()** para verificar se os campos obrigatórios do formulário estão preenchidos. Se essa validação falhar (retornando **false**), a submissão do formulário é impedida através de **event.preventDefault()**.
- **Comparação de Valores:** Posteriormente, outra função chamada **compararValores()** é executada para verificar se determinados valores do formulário atendem a certos critérios. Se essa comparação falhar (retornando **false**), a submissão do formulário também é impedida.

Essa organização é feita para garantir que a interface do usuário esteja limpa de mensagens de erro antigas e que quaisquer erros de validação sejam tratados antes que o formulário seja realmente submetido.

Nesta aula, vamos aprender a simular o envio de dados de um formulário utilizando o objeto **FormData** do JavaScript. Antes de explorarmos esse conceito, vamos iniciar nosso formulário com a estrutura HTML abaixo:

```
1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="UTF-8" />
5      <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6      <title>Envio de Dados em Formulários</title>
7      <style>
8        label {
9          display: block;
10         margin-bottom: 8px;
11       }
12     </style>
13   </head>
14   <body>
15     <h2>Envio de Dados em Formulários</h2>
16
17     <form id="meuFormulario">
18       <label for="nome">Nome:</label>
19       <input type="text" id="nome" name="nome" />
20
21       <label for="email">E-mail:</label>
22       <input type="email" id="email" name="email" />
23
24       <br />
25
26       <input type="submit" value="Enviar" />
27     </form>
28     <script src=".//script.js"></script>
29   </body>
30 </html>
```

Envio de Dados em Formulários

Nome:

E-mail:

Enviar

Em JavaScript, um objeto é uma estrutura de dados que permite armazenar várias informações relacionadas juntas. Ele é composto por pares de "chave" e "valor", onde cada chave é única dentro do objeto. As chaves são também conhecidas como propriedades do objeto, e seus valores podem ser de qualquer tipo de dado, incluindo números, strings, booleanos, arrays, outros objetos e até mesmo funções.

A estrutura básica de um objeto em JavaScript é definida entre chaves {}, onde as propriedades são especificadas no formato **chave: valor**, separadas por vírgulas.

```
const objeto = { chave: "valor", placa: "valor", portas: 4 };
```

Podemos fazer uma analogia dos objetos em JavaScript com coisas tangíveis da vida real para ajudar a entender melhor. Imagine um objeto em JavaScript como uma caixa de ferramentas. Dentro dessa caixa, você pode ter diferentes tipos de ferramentas, como chaves de fenda, martelos, alicates e assim por diante. Cada ferramenta tem sua própria utilidade e características específicas.

Da mesma forma, um objeto em JavaScript contém diferentes "propriedades", que são como as ferramentas na caixa. Cada propriedade tem um nome que a identifica, assim como cada ferramenta na caixa tem uma função específica. Por exemplo, em um objeto que representa uma pessoa, você pode ter propriedades como "nome", "idade", "cidade", que são como informações sobre a pessoa.

Assim como você pode acessar uma ferramenta específica na caixa de ferramentas quando precisa dela, você pode acessar os valores das propriedades de um objeto em JavaScript quando precisa das informações que elas representam.

FormData em JavaScript é um objeto que permite a criação de conjuntos de dados a serem enviados via requisições HTTP, especialmente útil em requisições do tipo POST. Ele é comumente usado em conjunto com a API **fetch()** ou em solicitações de formulários HTML.

- **Objeto FormData:** O **FormData** é um objeto incorporado ao JavaScript que permite construir facilmente um conjunto de pares chave/valor representando campos de um formulário HTML. Isso pode incluir campos de entrada de texto, caixas de seleção, botões de rádio, arquivos, etc.
- **Criação de Dados para Envio:** O objetivo principal do **FormData** é coletar dados do usuário e prepará-los para serem enviados ao servidor web. Isso é particularmente útil em situações onde você precisa enviar dados de um formulário sem recarregar a página.
- **Facilita o Envio de Dados:** Ao usar o **FormData**, você não precisa criar manualmente uma string de consulta (query string) para incluir nos parâmetros de uma requisição POST. O objeto **FormData** faz isso automaticamente para você.
- **Suporte a Envio de Arquivos:** Uma característica poderosa do **FormData** é que ele suporta o envio de arquivos junto com os dados do formulário. Isso é feito incluindo objetos de arquivo no conjunto de dados.
- **Integração com a API Fetch:** O **FormData** é frequentemente utilizado em conjunto com a API **fetch()**, que é uma maneira moderna e poderosa de fazer requisições HTTP no JavaScript. Você pode passar um objeto **FormData** como corpo da requisição em uma chamada fetch para enviar os dados para o servidor.
- **Compatibilidade com Formulários HTML:** O **FormData** pode ser facilmente criado a partir de um formulário HTML existente. Basta selecionar o formulário usando JavaScript e passá-lo para o construtor do **FormData**. Isso é útil quando você deseja capturar todos os dados de um formulário de uma vez.

O código JavaScript abaixo é um exemplo de como usar o **FormData** para capturar os dados de um formulário HTML e exibi-los no console.

```
document.addEventListener("DOMContentLoaded", function () {
  const formulario = document.getElementById("meuFormulario");

  formulario.addEventListener("submit", function (event) {
    event.preventDefault(); //impede a submissão tradicional do formulário

    let dados = new FormData(formulario);

    //exibir os dados no console
    for (let [chave, valor] of dados.entries()) {
      console.log(chave + ":" + valor);
    }
  });
});
```

Aqui está uma explicação linha por linha, enfatizando o uso do **FormData**:

- **formulario.addEventListener("submit", function (event) {**: Este trecho adiciona um ouvinte de evento ao formulário HTML. Ele escuta o evento de submissão (**submit**) do formulário e executa uma função quando esse evento ocorre.
- **event.preventDefault();**: Esta linha impede o comportamento padrão de submissão do formulário, que geralmente resultaria na recarga da página. Em vez disso, vamos lidar com a submissão do formulário de forma assíncrona usando JavaScript.

- **let dados = new FormData(formulario);**: Aqui, estamos criando um novo objeto **FormData**. Passamos o formulário como argumento para o construtor **FormData**, o que permite que o **FormData** cole automaticamente os dados de todos os campos do formulário.
- **for (let [chave,valor] of dados.entries()) {**: Este é um loop **for...of** que itera sobre todos os pares chave/valor dentro do objeto **FormData**. O método **entries()** retorna um iterador com todos os pares de chave/valor.
- **console.log(chave + ":" + valor);**: Dentro do loop, estamos utilizando **console.log()** para imprimir cada par chave/valor no console do navegador. A chave representa o nome do campo do formulário e o valor é o valor inserido pelo usuário nesse campo.

Em resumo, este código usa o **FormData** para capturar os dados de um formulário HTML, impedindo a submissão tradicional do formulário. Ele então itera sobre esses dados usando um loop **for...of**, exibindo cada par chave/valor no console. Isso é útil para depurar e verificar os dados que estão sendo enviados antes de enviar o formulário para o servidor.

Envio de Dados em Formulários

Nome:

E-mail:

Console output:

```
nome:Millene
email:email@email.com
```

script.js:13

script.js:13

O objeto "Window" é uma peça central na programação JavaScript em ambientes de navegador web. Ele representa a janela do navegador e oferece uma interface para manipulação e controle da janela e seu conteúdo. Aqui estão algumas das suas principais funcionalidades:

- **Controle da Janela:** O objeto "Window" gerencia propriedades da janela do navegador, como tamanho, posição e estado (por exemplo, maximizado, minimizado).
- **Acesso ao Documento:** Através do objeto "Window", é possível acessar o documento HTML carregado na janela, permitindo manipulação de elementos, estilos, eventos e outros aspectos do DOM (Document Object Model).
- **Navegação:** O objeto "Window" possibilita a navegação do usuário, permitindo redirecionamento para novas URLs, abertura de novas janelas ou abas e controle de histórico de navegação.
- **Temporizadores:** É possível agendar a execução de tarefas em intervalos específicos ou após um certo tempo usando temporizadores fornecidos pelo objeto "Window".
- **Eventos:** Gerencia eventos relacionados à janela e à interação do usuário, como eventos de carregamento da página, cliques do mouse e pressionamentos de tecla.

Além disso, o objeto "Window" está intimamente ligado à estrutura do Document Object Model (DOM) no navegador. O DOM é uma representação hierárquica dos elementos HTML de uma página da web. O objeto "Window" oferece acesso e controle sobre essa estrutura DOM, permitindo manipulação dinâmica do conteúdo da página.

Em resumo, o objeto "Window" desempenha um papel fundamental na programação JavaScript para web, fornecendo controle sobre a janela do navegador e interação com o conteúdo da página através do DOM. Ele capacita os desenvolvedores a criar experiências interativas e responsivas para os usuários na web.

O objeto **window** em JavaScript é uma estrutura fundamental que oferece uma variedade de propriedades para interação e manipulação do navegador. Aqui estão algumas das propriedades mais importantes desse objeto:

- **Document**: Esta propriedade representa o documento carregado na janela do navegador, proporcionando acesso ao DOM (Modelo de Objeto de Documento). Com isso, é possível dinamicamente manipular o conteúdo HTML, como adicionar, modificar ou remover elementos da página.
- **Location**: Outra propriedade relevante é **Location**, que fornece informações detalhadas sobre o URL da página atual. Isso inclui o host, o caminho, a consulta (query) e o fragmento (hash) da URL, permitindo a manipulação ou o acesso a partes específicas do URL conforme necessário.
- **Console**: A propriedade **console** é uma ferramenta crucial para depurar código JavaScript. Ela representa a janela do console do navegador, possibilitando exibir mensagens de log, erros, avisos e objetos, auxiliando no desenvolvimento e na identificação de problemas.
- **History**: Com a propriedade **History**, é possível navegar para frente e para trás no histórico do navegador. Métodos como **back()** e **forward()** permitem controlar o histórico de navegação do usuário de forma programática.
- **Screen**: Esta propriedade oferece informações sobre a tela do dispositivo, incluindo largura, altura, resolução e profundidade de cores. Isso é útil para adaptar aplicações a diferentes tamanhos de tela e dispositivos.
- **InnerWidth e InnerHeight**: Essas propriedades representam a largura e altura da área de conteúdo da janela do navegador, excluindo barras de rolagem, bordas e outros elementos de interface do usuário. Elas são úteis ao criar layouts responsivos ou determinar o espaço disponível para exibir conteúdo dentro da janela do navegador.

Vamos agora utilizar essas propriedades e visualizar em uma página como podemos acessá-las e o que elas retornam.

Ao desenvolver uma página web, podemos acessar as propriedades do objeto **window** através do JavaScript incorporado na página. Por exemplo, para acessar a propriedade **document**, podemos usar **window.document**. Isso nos dá acesso ao documento HTML carregado na janela do navegador.



Da mesma forma, para acessar a propriedade **location**, utilizamos **window.location**. Isso nos permite obter informações sobre o URL da página atual, como o host, o caminho e outros componentes do URL.



Além disso, podemos acessar e modificar o atributo **href** de **window.location** para alterar dinamicamente o URL da página sem a necessidade de recarregar a página. Por exemplo, para obter o URL completo da página atual, podemos usar **window.location.href**. Da mesma forma, para alterar o URL da página para uma nova URL, podemos simplesmente atribuir uma nova string à propriedade **href**.



```
> console.log(window.location.href);
https://www.hashtagtreinamentos.com/
```

Para obter informações sobre a tela do dispositivo, usamos **window.screen**, que nos fornece informações como a largura, altura e resolução da tela.

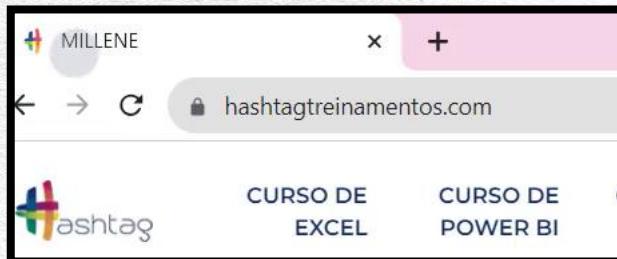
```
> console.log(window.screen.width);
1536
< undefined
VM1374:1
```

E, para obter a largura e altura da área de conteúdo da janela do navegador, usamos as propriedades **innerWidth** e **innerHeight** de **window**.

```
> console.log(window.innerWidth);
843
VM1468:1
```

Ao acessar essas propriedades, podemos utilizá-las para diversos fins, como ajustar o layout da página, extrair informações sobre o ambiente de visualização do usuário e depurar o código JavaScript.

```
> window.document.title = "MILLENE";
< 'MILLENE'
>
```



Quando você faz alterações no Console do DevTools em relação ao objeto **window**, como modificando propriedades como **location.href**, você está interagindo com a página web carregada no navegador apenas temporariamente durante a sessão atual do navegador. Essas alterações são visíveis apenas para você, o usuário atual, e não são permanentes.

Se você modificar a propriedade **location.href**, por exemplo, você pode simular a navegação para uma nova página, mas isso não afeta a versão real do site para outros usuários ou para você mesmo em sessões futuras. Assim que a página for recarregada ou você fechar o navegador, todas as modificações feitas no DevTools serão perdidas e a página voltará ao seu estado original.

Portanto, embora seja útil para testar alterações temporárias ou depurar código, as alterações feitas no DevTools não representam uma modificação permanente no site. Para fazer alterações permanentes, você precisaria acessar e modificar o código-fonte real do site.

Em desenvolvimento web, a capacidade de armazenar dados de forma eficiente é crucial. JavaScript oferece várias opções para lidar com isso, desde armazenamento local no navegador até comunicação com bancos de dados externos. Vamos explorar os conceitos básicos e os métodos mais comuns de armazenamento de dados em JavaScript:

- **LocalStorage e SessionStorage:**

- O LocalStorage e o SessionStorage são métodos de armazenamento de dados no navegador. O LocalStorage permite armazenar dados de forma persistente, permanecendo disponíveis mesmo após o fechamento e reabertura do navegador. Já o SessionStorage mantém os dados apenas durante a sessão atual do navegador, sendo descartados quando o navegador é fechado.

- **Cookies:**

- Os cookies são pequenos arquivos de texto armazenados no navegador do usuário. Eles são amplamente utilizados para armazenar informações como preferências do usuário, identificação de sessão e rastreamento de usuários. Apesar de suas limitações em termos de tamanho e desempenho, os cookies ainda são uma opção válida para armazenamento de dados.

- **IndexedDB:**

- IndexedDB é uma API poderosa para armazenamento de dados no navegador. Ele permite armazenar grandes quantidades de dados de forma estruturada e indexada, facilitando consultas eficientes. IndexedDB é ideal para aplicativos da web que precisam de armazenamento offline ou lidam com grandes conjuntos de dados.

- **Bancos de Dados Externos:**

- Além dos métodos de armazenamento local no navegador, é possível também armazenar dados em bancos de dados externos. Isso geralmente é feito por meio de APIs RESTful ou outras interfaces de comunicação com servidores. Bancos de dados externos oferecem maior escalabilidade e capacidade de gerenciamento de dados, mas exigem uma conexão com a internet e uma infraestrutura de servidor.

Esses métodos de armazenamento de dados em JavaScript oferecem opções flexíveis para lidar com diferentes requisitos de armazenamento, desde pequenas configurações de aplicativos até grandes conjuntos de dados. Entender esses conceitos básicos é fundamental para o desenvolvimento eficaz de aplicativos da web.



Você pode verificar esses armazenamentos usando as ferramentas de desenvolvedor (DevTools) disponíveis nos navegadores modernos, como Google Chrome, Mozilla Firefox, Microsoft Edge, entre outros. Abaixo estão as etapas para acessar cada um desses armazenamentos:

- **LocalStorage e SessionStorage:**

- No Google Chrome, por exemplo, você pode acessar o LocalStorage e o SessionStorage indo para as ferramentas de desenvolvedor (pressionando F12 ou clicando com o botão direito do mouse e selecionando "Inspecionar"), e em seguida, indo para a aba "Application" (ou "Aplicação" em português, dependendo do idioma do navegador). Dentro da aba "Application", você encontrará as seções "LocalStorage" e "SessionStorage" no painel esquerdo. Lá, você pode ver os itens armazenados, seus valores e até mesmo excluí-los, se necessário.

- **Cookies:**

- Os cookies também podem ser encontrados na aba "Application" nas ferramentas de desenvolvedor. Basta expandir a seção "Cookies" para ver todos os cookies associados ao site em questão. Aqui, você pode ver os detalhes de cada cookie, como nome, valor, domínio, data de expiração, entre outros.

- **IndexedDB:**

- Para IndexedDB, a visualização é um pouco mais complexa. Você pode encontrá-la na mesma aba "Application", mas dentro da seção "IndexedDB". Lá, você verá todos os bancos de dados indexados disponíveis para o site em questão. Você pode expandir cada banco de dados para ver suas respectivas coleções e até mesmo executar consultas dentro do DevTools.

- **Bancos de Dados Externos:**

- Bancos de dados externos não são diretamente acessíveis através das ferramentas de desenvolvedor do navegador, pois eles residem em servidores remotos. No entanto, você pode monitorar as solicitações de rede na aba "Network" (ou "Rede" em português) para ver as interações do navegador com esses bancos de dados externos. Isso pode incluir solicitações HTTP para APIs RESTful ou outras formas de comunicação com o servidor.

Módulo 9 – Introdução ao Armazenamentos de Dados (3 / 3)

720

The screenshot shows the Microsoft Edge DevTools interface. The main window has tabs for Application, Console, Sources, Network, and Application (which is currently selected). The Application tab displays storage information for the URL `https://www.bing.com`. Under the 'Storage' section, 'Session storage' is highlighted. A modal window titled 'Session storage' provides more details about session storage, including a 'Learn more' link.

The Network tab is also visible, showing a timeline with intervals of 20 ms, 40 ms, 60 ms, 80 ms, and 100 ms. There are several network requests listed, with the first one being a 'Fetch/XHR' request to `https://www.bing.com`.

The left sidebar of the DevTools shows the 'Aplicativo' (Application) panel with sections for Manifest, Service workers, and Armazenamento (Storage). The Storage section is expanded, showing 'Armazenamento local' (LocalStorage) and 'Armazenamento de sessão' (SessionStorage). The SessionStorage entry for `https://www.bing.com` is selected, displaying key-value pairs such as `bingi#1` and `bingIndex`.



O **LocalStorage** é uma ferramenta fundamental no desenvolvimento web, permitindo que os navegadores armazenem dados de forma persistente no dispositivo do usuário. Sua estrutura básica é simplificada e altamente funcional, operando com o conceito chave-valor, escopo de domínio e origem, limite de armazenamento, persistência e uma API simples.

Chave-Valor: A estrutura do LocalStorage é baseada no princípio chave-valor. Isso significa que os dados são armazenados em pares, onde uma chave única é associada a um valor correspondente. Essa abordagem simplifica o processo de armazenamento e recuperação de informações, permitindo acesso rápido e eficiente aos dados.

Storage	Key	Value
▼ Local storage	lastExternalReferrerTime	1709620077515
https://player.vimeo.com	elementor	{"__expiration":0,"pageViews":19,"sessions":14,"p...
https://www.hashtagtreinamen	lastExternalReferrer	empty
▶ Session storage	li_adsId	17495b04-86f7-49bc-a77b-b069b34da853

Semelhança com Dados Tipo Objeto: Embora o LocalStorage armazene dados em pares chave-valor, é importante notar que ele só pode armazenar valores como strings. Isso significa que, mesmo que você deseje armazenar um objeto JavaScript no LocalStorage, será necessário converter esse objeto em uma string antes de armazená-lo. Essa semelhança com dados tipo objeto não é direta, mas é comum a prática de converter objetos em strings usando a função **JSON.stringify()** antes de armazená-los e depois convertê-los de volta para objetos usando **JSON.parse()** ao recuperá-los.

```
> localStorage.getItem("lastExternalReferrerTime");
< '1709620077515'
```



Escopo de Domínio e Origem: O LocalStorage opera dentro do escopo de domínio e origem do navegador. Isso significa que os dados armazenados em LocalStorage estão associados a um domínio específico e ao protocolo de origem (HTTP ou HTTPS) do site que os criou. Essa limitação garante que os dados só possam ser acessados pelo mesmo site que os criou, proporcionando segurança e privacidade aos usuários.

Limite de Armazenamento: Embora o LocalStorage seja uma ferramenta poderosa, há limites para a quantidade de dados que pode armazenar. Os navegadores geralmente impõem um limite de armazenamento para o LocalStorage, variando de alguns megabytes a várias dezenas de megabytes, dependendo do navegador e da configuração do dispositivo. É importante estar ciente desses limites ao projetar aplicativos que fazem uso intensivo do armazenamento local.

Persistência: Uma das características distintivas do LocalStorage é sua capacidade de persistir dados mesmo após o fechamento do navegador ou a reinicialização do dispositivo. Isso significa que os dados armazenados permanecem disponíveis para acesso futuro, proporcionando uma experiência consistente para o usuário. No entanto, é importante notar que os dados armazenados em LocalStorage não são transferidos automaticamente entre dispositivos ou navegadores.

API Simples: O LocalStorage oferece uma API simples e intuitiva para manipular dados. Com apenas alguns métodos básicos, como **setItem**, **getItem**, **removeItem** e **clear**, os desenvolvedores podem facilmente adicionar, recuperar e excluir dados do armazenamento local. Essa simplicidade torna o LocalStorage uma escolha popular para armazenar informações como preferências do usuário, estado da aplicação e dados de cache.

Se você digitar "localStorage" no console do DevTools de um navegador da web, você acessará o objeto **localStorage**, que é uma parte do objeto **window** no ambiente do navegador. O **localStorage** é um recurso do navegador que permite armazenar dados no dispositivo do usuário de forma persistente entre sessões e recargas da página.

Ao digitar "localStorage" e pressionar Enter no console, você verá uma representação do objeto **localStorage** no console, mostrando todos os métodos e propriedades disponíveis para interação. Isso geralmente inclui métodos como **setItem()**, **getItem()**, **removeItem()** e **clear()**, que são usados para adicionar, recuperar, remover e limpar itens armazenados no **localStorage**.

Você pode interagir com o **localStorage** diretamente no console, executando comandos como **localStorage.setItem('chave', 'valor')** para adicionar um item, **localStorage.getItem('chave')** para recuperar um item, **localStorage.removeItem('chave')** para remover um item específico ou **localStorage.clear()** para limpar todos os itens armazenados no **localStorage**.

Esta é uma maneira útil de testar e depurar o uso do **localStorage** em seu código JavaScript durante o desenvolvimento de aplicativos web.

```
> localStorage
< Storage {lastExternalReferrerTime: '1709620077515', elementor: '{"_expiration":{}, "pageViews": 19, "sessions":14, "popup_13160_times":14}', lastExternalReferrer: 'empty', li_adsId: '17495b04-86f7-49bc-a77b-b069b34da853', length: 4} i
  elementor: {"_expiration":{}, "pageViews":19, "sessions":14, "popup_13160_times":14}
  lastExternalReferrer: "empty"
  lastExternalReferrerTime: "1709620077515"
  li_adsId: "17495b04-86f7-49bc-a77b-b069b34da853"
  length: 4
  [[Prototype]]: Storage
    > clear: f clear()
    > getItem: f getItem()
    > key: f key()
    > length: (...)

    > removeItem: f removeItem()
    > setItem: f setItem()
    > constructor: f Storage()
    Symbol(Symbol.toStringTag): "Storage"
    > get length: f length()
    > [[Prototype]]: Object
```

Nesta aula, vamos aprender alguns métodos do localStorage e sua aplicação em um formulário. Para isso, vamos iniciar um arquivo index.html com a estrutura abaixo. E já deixar aberto o DevTools na aba de Aplicações > localStorage.

```
1  <!DOCTYPE html>
2  <html lang="en">
3      <head>
4          <meta charset="UTF-8" />
5          <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6          <title>LocalStorage</title>
7      </head>
8      <body>
9          <h1>Armazenamento Local</h1>
10         <form id="form">
11             <label for="nome">Nome:</label>
12             <input type="text" id="nome" /><br /><br />
13             <label for="idade">Idade:</label>
14             <input type="number" id="idade" /><br /><br />
15
16             <button type="submit">Salvar</button>
17         </form>
18         <br />
19         <button id="limpar">Limpar Dados</button>
20         <script src="script.js"></script>
21     </body>
22 </html>
```

Armazenamento Local

Nome:

Idade:

Application

- Session storage
- IndexedDB
- Web SQL
- Cookies
- Private state tokens
- Interest groups
- Cloud storage

Key	Value

Agora no arquivo script.js vamos implementar os conceitos que aprendemos durante este módulo de JavaScript. Vamos dar uma olhada neste código para entender como podemos interagir com elementos HTML e capturar dados de um formulário.

```
document.addEventListener("DOMContentLoaded", function () {
  const form = document.getElementById("form");
  const limparBotao = document.getElementById("limpar");

  form.addEventListener("submit", function(event) {
    event.preventDefault();

    // Capturar os dados do formulário
    const nome = document.getElementById("nome").value;
    const idade = document.getElementById("idade").value;

  })
});
```

Primeiro, estamos usando o evento "DOMContentLoaded". Isso significa que estamos esperando que a página carregue completamente antes de executar nosso código. Isso é importante para garantir que todos os elementos da página estejam disponíveis para interação.

Em seguida, estamos capturando elementos HTML importantes, como o formulário e um botão de limpar. Esses elementos são armazenados em variáveis para que possamos usá-los mais tarde.

Agora, vamos focar no evento de envio do formulário. Quando o formulário é enviado, queremos processar os dados sem recarregar a página. Para fazer isso, usamos **event.preventDefault()**. Isso evita o comportamento padrão do formulário, que seria enviar os dados e recarregar a página. Em vez disso, podemos manipular os dados sem interrupções.

Depois de prevenir o comportamento padrão, capturamos os dados do formulário. Estamos obtendo o nome e a idade que o usuário inseriu nos campos do formulário. Isso é feito usando o método **getElementById().value**. Esses dados são então armazenados em variáveis para uso posterior.



localStorage.setItem():

- O **localStorage** é um recurso do navegador que permite armazenar dados de forma persistente no navegador do usuário.
- O método **setItem()** do **localStorage** é usado para armazenar um par chave-valor. Ele aceita dois argumentos: o nome da chave e o valor a ser armazenado. Por exemplo, **localStorage.setItem("nome", "João")** armazenaria a string "João" sob a chave "nome" no **localStorage**.
- Se a chave já existir, o método **setItem()** atualiza o valor associado à chave. Caso contrário, cria uma nova entrada no **localStorage**.

```
//Verifica se os campos estão preenchidos
if (nome && idade) {
    //Salvo no localStorage
    localStorage.setItem("nome", nome);
    localStorage.setItem("idade", idade);

    alert("Dados salvos com sucesso!");
}
```

Key	Value
nome	Millene
idade	34

Explicação do Código:

- O código verifica se as variáveis **nome** e **idade** têm algum valor. Se ambos os campos estiverem preenchidos, o código dentro do bloco **if** será executado.
- Dentro do bloco **if**, os dados inseridos pelo usuário (nome e idade) são armazenados no **localStorage** usando o método **setItem()**.
 - O nome é armazenado com a chave "nome" e o valor é o conteúdo da variável **nome**.
 - A idade é armazenada com a chave "idade" e o valor é o conteúdo da variável **idade**.
- Após salvar os dados, um alerta é exibido para informar ao usuário que os dados foram salvos com sucesso.

Explicação do Código:

- Essas linhas de código têm a finalidade de limpar os campos de um formulário após determinada ação ou evento.
- O método **document.getElementById()** é usado para selecionar elementos HTML com base em seu ID. Ele retorna uma referência ao elemento com o ID especificado.
- No código fornecido, são selecionados os elementos de input do formulário com IDs "nome" e "idade".
- Em seguida, é atribuído um valor vazio "" para a propriedade **value** desses elementos.
 - Isso significa que o conteúdo dos campos de entrada é substituído por uma string vazia, removendo assim qualquer texto anteriormente inserido pelo usuário.
- Como resultado, após a execução dessas linhas de código, os campos de entrada do formulário ficarão vazios, prontos para receber novos dados do usuário.

Essas linhas são úteis quando você deseja fornecer ao usuário a opção de limpar os campos do formulário facilmente, por exemplo, após enviar com sucesso os dados ou antes de preencher novos dados. Isso ajuda a melhorar a experiência do usuário, tornando a interação com o formulário mais eficiente e intuitiva.

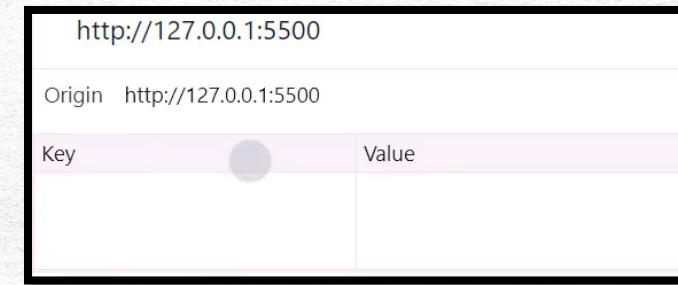
```
document.addEventListener("DOMContentLoaded", function () {  
    const form = document.getElementById("form");  
    const limparBotao = document.getElementById("limpar");  
  
    form.addEventListener("submit", function(event) {  
        event.preventDefault();  
  
        // Capturar os dados do formulário  
        const nome = document.getElementById("nome").value;  
        const idade = document.getElementById("idade").value;  
  
        // Verifica se os campos estão preenchidos  
        if (nome && idade) {  
            // Salvo no localStorage  
            localStorage.setItem("nome", nome);  
            localStorage.setItem("idade", idade);  
  
            // Limpar os campos do formulário  
            document.getElementById("nome").value = "";  
            document.getElementById("idade").value = "";  
  
            alert("Dados salvos com sucesso!");  
        } else {  
            console.log("Por favor, preencha todos os campos");  
        }  
    });  
});
```



O método **removeItem** do objeto **localStorage** em JavaScript é utilizado para remover um item específico do armazenamento local do navegador. O armazenamento local fornece uma maneira simples de armazenar dados no navegador de forma persistente entre sessões, sendo útil para armazenar informações como preferências do usuário, configurações e dados de cache.

O **removeItem** requer apenas um parâmetro, que é a chave (ou identificador) do item que se deseja remover. Ao chamar este método com a chave do item desejado, o item correspondente é removido do armazenamento local. Se o item não existir, o **removeItem** não fará nada.

```
// Limpar localStorage - removeItem()
limparBotao.addEventListener("click", function () {
    localStorage.removeItem("nome");
    localStorage.removeItem("idade");
    alert("Dados foram removidos do localStorage");
});
```



- **function() { localStorage.removeItem('nome'); localStorage.removeItem('idade'); }**

Esta é a função que será executada quando o botão for clicado. Aqui está o que está acontecendo dentro dela:

- **localStorage.removeItem('nome');**: Isso remove um item chamado '**nome**' do armazenamento local. Se existir um item com a chave '**nome**', ele será removido. Se não existir, nada acontecerá.
- **localStorage.removeItem('idade');**: Isso remove um item chamado '**idade**' do armazenamento local. Da mesma forma, se existir um item com a chave '**idade**', ele será removido. Se não existir, nada acontecerá.

Resumindo, quando o botão associado a **limparBotao** for clicado, esta função será acionada e removerá dois itens, '**nome**' e '**idade**', do armazenamento local, se eles existirem. Isso pode ser útil em situações em que você deseja limpar certos dados armazenados localmente no navegador, como em um aplicativo da web.

Vamos adicionar mais uma botão dentro do arquivo HTML, chamado "Carregar" e também um elemento de um parágrafo vazio, que receberá as informações que vamos carregar.

```
<button id="carregar">Carregar</button>
<p id="texto"></p>
```

O método **getItem** do objeto **localStorage** em JavaScript é usado para recuperar o valor associado a uma chave específica armazenada no armazenamento local do navegador. O armazenamento local é uma forma de armazenamento persistente no navegador da web, que permite que os desenvolvedores armazenem dados localmente no computador do usuário.

O **getItem** aceita um parâmetro, que é a chave do item que você deseja recuperar. Ao chamar este método com a chave desejada, ele retorna o valor associado a essa chave no armazenamento local, se existir. Se não houver nenhum valor associado à chave especificada, o método retornará **null**.

O nome do usuário é null e sua idade é null

```
// getItem - acessar os dados do nosso localStorage
const carregarBtn = document.getElementById("carregar");
const texto = document.getElementById("texto");

carregarBtn.addEventListener("click", function () {
    const dadoNome = localStorage.getItem("nome");
    const dadoIdade = localStorage.getItem("idade");

    texto.textContent = `O nome do usuário é ${dadoNome} e sua idade é ${dadoIdade}`;
});
```

- **function() { const dadoNome = localStorage.getItem('nome'); const dadoidade = localStorage.getItem('idade'); texto.textContent = O nome do usuário é \${dadoNome} e a sua idade é \${dadoidade}; }**

Esta é a função que será executada quando o botão for clicado. Aqui está o que está acontecendo dentro dela:

- **const dadoNome = localStorage.getItem('nome');**: Isso recupera o valor associado à chave '**nome**' do armazenamento local e o armazena na variável **dadosNome**.
- **const dadoidade = localStorage.getItem('idade');**: Isso recupera o valor associado à chave '**idade**' do armazenamento local e o armazena na variável **dadosIdade**.
- **texto.textContent = `O nome do usuário é \${dadoNome} e a sua idade é \${dadoidade}`;**: Aqui, estamos atualizando o conteúdo do elemento HTML com o ID **texto**. O texto deste elemento será definido como uma string que inclui os valores recuperados do armazenamento local. O **\${dadoNome}** e **\${dadoidade}** são substituídos pelos valores recuperados das chaves '**nome**' e '**idade**', respectivamente.

Resumindo, quando o botão associado a **carregarBtn** é clicado, esta função é acionada. Ela recupera os valores associados às chaves '**nome**' e '**idade**' do armazenamento local e atualiza o conteúdo de um elemento HTML com esses valores. Isso geralmente é usado para exibir os dados armazenados localmente na interface do usuário de uma aplicação web.

Módulo 10

PROJETO: HASHFORMS

PROJETO: HASHFORMS

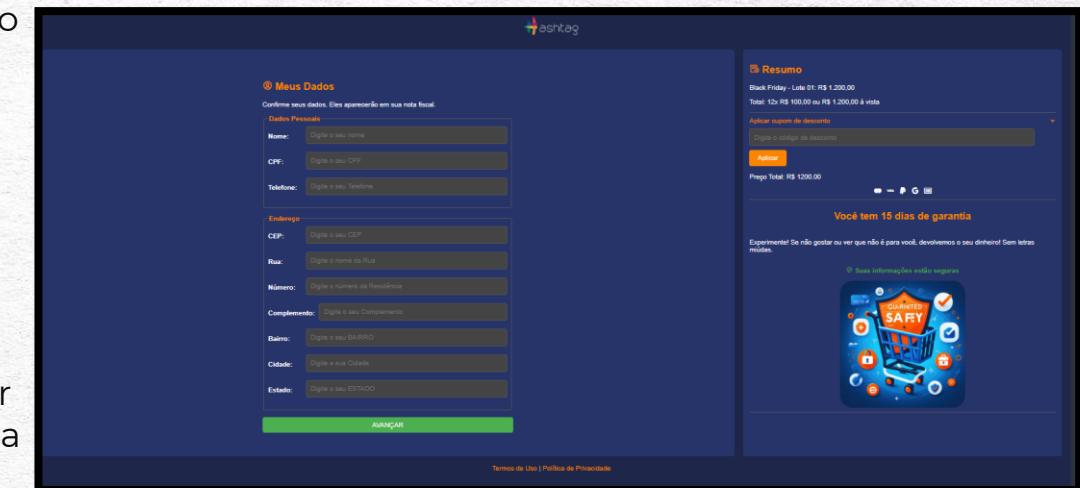
PROJETO: HASHFORMS



Neste projeto, vamos desenvolver um sistema de formulário que será responsável por capturar e validar dados adicionados pelos usuários. Nossa abordagem incluirá as seguintes funcionalidades principais:

- Formulário:** Criação de um formulário interativo onde os usuários poderão inserir seus dados. Será essencial garantir que todos os campos necessários sejam preenchidos de forma correta e eficiente.
- Validação de Dados:** Implementaremos mecanismos de validação para garantir que os dados inseridos pelos usuários sejam precisos e consistentes. Isso inclui verificações de formato, como emails e números de telefone.
- Aplicação de Desconto:** Incorporaremos uma funcionalidade para aplicar descontos nas compras. Para isso, utilizaremos um toggle que expandirá a área destinada ao desconto, permitindo aos usuários inserir códigos promocionais.
- Armazenamento no localStorage:** Para proporcionar uma melhor experiência ao usuário, o valor do desconto será armazenado no localStorage. Dessa forma, mesmo que o usuário navegue por diferentes páginas ou reinicie o navegador, o desconto permanecerá aplicado.

Com essas funcionalidades, o projeto visa oferecer uma experiência de usuário fluida e eficiente, garantindo que todas as etapas, desde a inserção de dados até a aplicação de descontos, sejam realizadas de maneira simples e intuitiva.



Estrutura Básica do HTML

Vamos começar criando a estrutura básica do nosso documento HTML. Essa estrutura define a base para o conteúdo que será exibido na página.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Projeto HashForms</title>
    <link href="https://unpkg.com/boxicons@2.1.4/css/boxicons.min.css" rel="stylesheet" />
  </head>
  <body>
    <!-- Conteúdo aqui -->
  </body>
</html>
```

Adicionando Imagens

Em seguida, vamos adicionar as imagens que serão exibidas na página. As imagens são importantes para melhorar a estética e fornecer informações visuais.

```
<div class="images">
  
  
</div>
```

Explicação:

- Criamos uma `<div>` com a classe `images` para agrupar as imagens.
- Utilizamos o elemento `` e o atributo `src` para definir a URL da imagem que será exibida.

Criando o Formulário

Agora, vamos criar a estrutura do formulário onde os usuários poderão inserir seus dados pessoais. Vamos garantir que todos os campos necessários estejam presentes.

Explicação:

- Utilizamos a `<div>` com a classe `boxes` para agrupar os elementos do formulário.
- A `<div>` com a classe `box1` contém o título, a descrição e o formulário.
- O elemento `<form>` define o formulário e o atributo `action="#"` indica que ele será processado na mesma página.
- Utilizamos o elemento `fieldset` para agrupar os campos relacionados e o `legend` para adicionar uma legenda ao grupo.
- Cada campo do formulário é composto por um `label` e um `input`, sendo o `required` utilizado para garantir que os campos sejam preenchidos.

```
<div class="boxes">
  <div class="box1">
    <h1><i class="bx bx-user-circle"></i> Meus Dados</h1>
    <p>Confirme seus dados. Eles aparecerão em sua nota fiscal.</p>
    <form action="#" id="formulario">
      <fieldset>
        <legend>Dados Pessoais</legend>
        <div class="form-group">
          <label for="nome">Nome:</label>
          <input type="text" name="nome" id="nome" placeholder="Digite o seu nome" required />
        </div>
        <div class="form-group">
          <label for="cpf">CPF:</label>
          <input type="number" name="cpf" id="cpf" placeholder="Digite o seu CPF" required />
        </div>
        <div class="form-group">
          <label for="telefone">Telefone:</label>
          <input type="number" name="telefone" id="telefone" placeholder="Digite o seu Telefone" required />
        </div>
      </fieldset>
    </form>
  </div>
</div>
```

Conceitos de Validação de Dados

Para garantir que os dados inseridos pelos usuários sejam precisos e consistentes, implementamos a validação de dados.

Validação de Dados

- O atributo required nos elementos de entrada (input) garante que o campo seja preenchido antes do envio do formulário.
- Outros métodos de validação podem ser implementados com JavaScript para realizar verificações mais complexas, como validar o formato de um email ou verificar se o CPF possui o número correto de dígitos.



⑧ Meus Dados

Confirme seus dados. Eles aparecerão em sua nota fiscal.

Dados Pessoais

Nome:

CPF:

Telefone:

Resumo

Na aula de hoje, vamos expandir nosso formulário HTML para incluir uma seção de endereço, permitindo que os usuários insiram informações detalhadas sobre seu local de residência. Além disso, adicionaremos um botão de submissão para que os dados sejam enviados.

Adicionando Campos de Endereço

Vamos começar adicionando um novo fieldset para agrupar os campos de endereço no formulário. O fieldset nos ajuda a organizar visualmente os campos relacionados, tornando o formulário mais claro e fácil de entender.

```
<fieldset>
  <legend>Endereço</legend>
  <div class="form-group">
    <label for="cep">CEP:</label>
    <input
      type="number"
      name="cep"
      id="cep"
      placeholder="Digite o seu CEP"
      required
    />
  </div>

  <div class="form-group">
    <label for="rua">Rua:</label>
    <input
      type="text"
      name="rua"
      id="rua"
      placeholder="Digite o nome da Rua"
      required
    />
  </div>

  <div class="form-group">
    <label for="numero">Número:</label>
    <input
      type="number"
      name="numero"
      id="numero"
      placeholder="Digite o número da Residência"
      required
    />
  </div>
```

```
<div class="form-group">
  <label for="complemento">Complemento:</label>
  <input
    type="text"
    name="complemento"
    id="complemento"
    placeholder="Digite o seu Complemento"
    required
  />
</div>

<div class="form-group">
  <label for="bairro">Bairro:</label>
  <input
    type="text"
    name="bairro"
    id="bairro"
    placeholder="Digite o seu Bairro"
    required
  />
</div>

<div class="form-group">
  <label for="cidade">Cidade:</label>
  <input
    type="text"
    name="cidade"
    id="cidade"
    placeholder="Digite a sua Cidade"
    required
  />
</div>
```

```
<div class="form-group">
  <label for="estado">Estado:</label>
  <input
    type="text"
    name="estado"
    id="estado"
    placeholder="Digite o seu Estado"
    required
  />
</div>
</fieldset>
```



Explicação dos Campos

- **CEP**: Campo para o Código de Endereçamento Postal, fundamental para identificar a localização.
- **Rua**: Nome da rua onde a pessoa reside.
- **Número**: Número da residência na rua especificada.
- **Complemento**: Qualquer informação adicional, como bloco ou apartamento.
- **Bairro**: Bairro onde a residência está localizada.
- **Cidade**: Nome da cidade.
- **Estado**: Nome do estado.

Todos esses campos são obrigatórios (required) para garantir que informações completas sejam fornecidas.

Adicionando o Botão de Submissão

Finalmente, vamos adicionar um botão para que os usuários possam enviar o formulário depois de preenchê-lo.

```
<button type="submit">AVANÇAR</button>
```

Explicação do Botão

- **type="submit"**: Define que o botão será usado para submeter o formulário.
- **Texto "AVANÇAR"**: O texto no botão indica a ação que será realizada ao clicar nele.

Nesta aula, aprendemos a expandir nosso formulário HTML, adicionando uma seção completa de endereço e um botão de submissão. Essas melhorias tornam o formulário mais completo e funcional, permitindo capturar todas as informações necessárias dos usuários.

Adicionando a Seção de Resumo

Vamos começar adicionando a seção de resumo ao nosso projeto. Essa seção exibirá o resumo da compra, incluindo o valor total e as opções de pagamento.

```
<div class="resumo">
  <h2><i class="bx bx-receipt"></i> Resumo</h2>
  <p>Black Friday - Lote 01: R$ 1.200,00</p>
  <p>Total: 12x R$ 100,00 ou R$ 1.200,00 à vista</p>
</div>
```

Explicação:

- **Classe "resumo":** Utilizamos uma div com a classe resumo para agrupar os elementos relacionados ao resumo da compra.
- **Título:** O elemento `<h2>` exibe o título "Resumo" junto com um ícone de recibo.
- **Parágrafos:** Utilizamos os elementos `<p>` para exibir as informações da oferta de Black Friday e o total da compra, com opções de pagamento parcelado e à vista.

Adicionando a Seção de Cupom de Desconto

Agora, vamos adicionar a seção para aplicação do cupom de desconto. Esta seção permitirá que os usuários insiram um código de desconto e vejam a mensagem correspondente ao cupom aplicado.

```
<div class="apply-discount">
  <h3><i class="bx bx-gift"></i> Aplicar cupom de desconto</h3>
  <input type="text" id="cupom" placeholder="Digite o código de desconto" />
  <button type="button" onclick="aplicarDesconto()">Aplicar</button>
  <span id="mensagem-desconto"></span>
  <p>Preço Total: R$ 1.200,00</p>
</div>
```

Explicação:

- **Classe "apply-discount":** Utilizamos uma div com a classe apply-discount para agrupar os elementos relacionados ao cupom de desconto.
- **Título:** O elemento `<h3>` exibe o título "Aplicar cupom de desconto" junto com um ícone de presente.
- **Input para Cupom:** Utilizamos o elemento `<input>` para criar um campo de texto onde os usuários podem inserir o código de desconto. O atributo placeholder fornece uma dica visual sobre o que deve ser inserido.
- **Botão de Aplicação:** Utilizamos o elemento `<button>` com o atributo `type="button"` para criar um botão que, ao ser clicado, chama a função `aplicarDesconto()`.
- **Mensagem de Desconto:** O elemento `` com o `id="mensagem-desconto"` será usado para exibir a mensagem correspondente ao cupom aplicado (ex.: sucesso ou erro).
- **Preço Total:** Um parágrafo `<p>` para mostrar o preço total da compra, que poderá ser atualizado conforme o desconto aplicado.

Vamos adicionar classes e ids necessários, além de uma estilização inline. Nossa estrutura ficará da seguinte forma:

```
<div class="box2">
  <section class="resumo">
    <h2><i class="bx bx-receipt"></i> Resumo</h2>
    <div class="resumo-item">
      <p>Black Friday - Lote 01: R$ 1.200,00</p>
      <p>Total: 12x R$ 100,00 ou R$ 1.200,00 à vista</p>
    </div>

    <div class="apply-discount">
      <div class="apply-discount-header">
        <span>Aplicar cupom de desconto</span>
        <i id="toggel-icon" class="bx bxs-chevron-down"></i>
      </div>
      <div class="apply-discount-body" id="apply-discount-body">
        <input
          type="text"
          id="discount-code"
          placeholder="Digite o código de desconto"
        />
        <button type="button">Aplicar</button>
        <span style="display: block; margin-top: 10px; color: green">
          Mensagem de desconto
        </span>
        <p id="total-price">Preço Total: R$ 1200.00</p>
      </div>
    </div>
  </section>
</div>
```

The screenshot shows a user interface for data entry. At the top right is a logo consisting of four colored squares. Below it is a section titled "Meus Dados" with a subtitle "Confirme seus dados. Eles aparecerão em sua nota fiscal." It contains fields for "Nome", "CPF", and "Telefone". A "Endereço" section follows, with fields for "CEP", "Rua", "Número", "Complemento", "Bairro", "Cidade", and "Estado". A large "AVANÇAR" button is located below these fields. Below this is a "Resumo" section containing the text "Black Friday - Lote 01: R\$ 1.200,00", "Total: 12x R\$ 100,00 ou R\$ 1.200,00 à vista", a dropdown menu "Aplicar cupom de desconto" with the placeholder "Digite o código de desconto", and a "Aplicar" button. At the bottom of the summary section is the text "Preço Total: R\$ 1200.00".

Adicionando Ícones de Pagamento

Vamos começar adicionando os ícones de pagamento ao nosso projeto. Esses ícones representam as formas de pagamento aceitas e melhoram a experiência visual do usuário.

```
<div class="icons">
  <i class="bx bxl-mastercard"></i>
  <i class="bx bxl-visa"></i>
  <i class="bx bxl-paypal"></i>
  <i class="bx bxl-google"></i>
  <i class="bx bx-barcode"></i>
</div>
```

Explicação:

- **Classe "icons":** Utilizamos uma div com a classe icons para agrupar todos os ícones de pagamento.
- **Ícones:** Cada ícone é representado por um elemento *<i>* com uma classe específica da biblioteca Boxicons (bx). Isso inclui ícones para Mastercard, Visa, PayPal, Google Pay e código de barras.

Adicionando Seção de Garantia

Agora, vamos adicionar a seção de garantia, que fornece informações sobre a política de devolução e segurança das informações.

```
<div class="garantia">
  <div class="garantia_box">
    <h3>Você tem 15 dias de garantia</h3>
    <p>
      Experimente! Se não gostar ou ver que não é para você,
      devolvemos o seu dinheiro! Sem letras miúdas.
    </p>
    <p class="safe">
      <i class="bx bx-check-shield"></i>
      <strong>Suas informações estão seguras</strong>
    </p>
    
  </div>
</div>
```



Explicação:

- **Classe "garantia":** Utilizamos uma div com a classe garantia para agrupar as informações sobre a garantia e segurança.
- **Título e Parágrafos:** O elemento `<h3>` exibe o título "Você tem 15 dias de garantia". Os elementos `<p>` fornecem informações detalhadas sobre a política de devolução e a segurança das informações.
- **Ícone de Segurança:** Adicionamos um ícone de segurança representado por `<i class="bx bx-check-shield"></i>`.
- **Imagen de Segurança:** Uma imagem representando a segurança das informações, com o atributo alt descrevendo a imagem e o atributo width definindo a largura.

Adicionando Rodapé

Por fim, vamos adicionar o rodapé ao nosso projeto. O rodapé contém links para os Termos de Uso e a Política de Privacidade.

```
<footer>
  <p>
    <a href="#">Termos de Uso</a> |
    <a href="#">Política de Privacidade</a>
  </p>
  <!-- | --> Alt + 124 (teclado numérico) -->
</footer>
```

Explicação:

- **Elemento `<footer>`:** Utilizamos o elemento `<footer>` para definir o rodapé da página.
- **Parágrafo com Links:** Dentro do rodapé, utilizamos um parágrafo `<p>` contendo links `<a>` para os Termos de Uso e a Política de Privacidade. O símbolo `|` é utilizado como separador entre os links.

Nesta aula, vamos dar os primeiros passos na estilização da nossa página utilizando CSS. Vamos aplicar uma estilização geral para melhorar a aparência e a usabilidade da nossa aplicação. Utilizaremos variáveis CSS para definir um esquema de cores consistente e estilizar diversos elementos da nossa página.

Objetivo

- Aprender a aplicar estilos gerais usando CSS.
- Utilizar variáveis CSS (:root) para definir um esquema de cores.
- Estilizar elementos como o corpo da página, contêineres, imagens, caixas e títulos.

Passo 1: Definindo Variáveis CSS

Começaremos definindo variáveis CSS no seletor :root. Essas variáveis nos permitirão criar um esquema de cores consistente e fácil de manter.

```
:root {  
    --background: #30343f;  
    --primary: #1e2749;  
    --secondary: #273469;  
    --text-realce: #fb8500;  
    --text: #aaa;  
    --focus: #ff9f1c;  
    --safe: #45a049;  
    --input-background: #444;  
    --input-border: #555;  
    --button-background: #4caf50;  
    --white: #fff;  
}
```

Explicação:

- --background: Cor de fundo principal da página.
- --primary: Cor de fundo primária usada em contêineres.
- --secondary: Cor de fundo secundária usada em caixas.
- --text-realce: Cor utilizada para realçar textos importantes, como títulos.
- --text: Cor padrão do texto.
- --focus: Cor usada para elementos focados, como inputs.
- --safe: Cor usada para indicar segurança, como mensagens de sucesso.
- --input-background: Cor de fundo dos inputs.
- --input-border: Cor da borda dos inputs.
- --button-background: Cor de fundo dos botões.
- --white: Cor branca, usada para textos e outros elementos.



Passo 2: Estilizando o Corpo da Página

Vamos aplicar estilos ao corpo da página (body) para definir a fonte, cor de fundo, margens, padding e cor do texto.

```
body {  
  font-family: "Arial", sans-serif;  
  background-color: var(--background);  
  margin: 0;  
  padding: 0;  
  color: var(--white);  
}
```

Explicação:

- font-family: Define a família de fontes utilizada na página.
- background-color: Aplica a cor de fundo definida na variável --background.
- margin **e** padding: Remove a margem e o padding padrão do corpo.
- color: Define a cor do texto como branca, utilizando a variável --white.

Passo 3: Estilizando o Contêiner Principal

Agora, vamos estilizar o contêiner principal (.container) para centralizar seu conteúdo e aplicar estilos de fundo, padding e sombra.

```
.container {  
  width: 80%;  
  margin: 0 auto;  
  background-color: var(--primary);  
  padding: 20px;  
  box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);  
  border-radius: 8px;  
}
```

Explicação:

- width: Define a largura do contêiner como 80% da largura da página.
- margin: Centraliza o contêiner horizontalmente.
- background-color: Aplica a cor de fundo definida na variável --primary.
- padding: Adiciona espaço interno ao contêiner.
- box-shadow: Adiciona uma sombra suave ao contêiner.
- border-radius: Define bordas arredondadas para o contêiner.

Passo 4: Estilizando a Seção de Imagens

Vamos estilizar a seção de imagens (.images) para centralizar o conteúdo e adicionar espaçamento.

```
.images {  
    text-align: center;  
    margin-bottom: 20px;  
    padding: 10px;  
}
```

Explicação:

- text-align: Centraliza o conteúdo da seção.
- margin-bottom: Adiciona espaçamento inferior.
- padding: Adiciona espaço interno ao redor do conteúdo.

Passo 5: Estilizando as Caixas de Dados

Vamos estilizar as caixas (.boxes, .box1, .box2) para definir a disposição e a aparência dos elementos.

```
.boxes {  
    display: flex;  
    flex-wrap: wrap;  
    gap: 20px;  
}  
  
.box1,  
.box2 {  
    flex: 1;  
    background-color: var(--secondary);  
    padding: 15px;  
    border-radius: 5px;  
}  
  
.box2 {  
    max-width: 30%;  
}
```

Explicação:

- display: flex: Utiliza o modelo flexbox para dispor os elementos das caixas.
- flex-wrap: Permite que os elementos se ajustem automaticamente ao contêiner.
- gap: Adiciona espaçamento entre os elementos das caixas.
- flex: 1: Define a proporção de espaço que cada caixa ocupa.
- background-color: Aplica a cor de fundo definida na variável --secondary.
- padding: Adiciona espaço interno às caixas.
- border-radius: Define bordas arredondadas para as caixas.
- max-width: Define a largura máxima da segunda caixa (.box2).

Passo 6: Estilizando Títulos

Por fim, vamos estilizar os títulos (h1, h2, h3) para aplicar a cor de realce.

```
h1,  
h2,  
h3 {  
  color: var(--text-realce);  
}
```

Explicação:

- color: Define a cor dos títulos utilizando a variável --text-realce.

Conclusão

Nesta aula, aprendemos a aplicar estilizações gerais usando CSS para melhorar a aparência da nossa página. Utilizamos variáveis CSS para criar um esquema de cores consistente e aplicamos estilos a diversos elementos. Isso torna nossa aplicação mais atraente e fácil de usar.



Explicação das Unidades de Medida em CSS

PX (Pixels)

Pixels (PX) são uma unidade de medida fixa baseada no valor da tela. Cada pixel é um ponto na tela, e seu tamanho pode variar dependendo da resolução do dispositivo.

Características:

- Unidade fixa e absoluta.
- Usada para medidas precisas.
- Não escala em resposta ao tamanho da fonte ou outras propriedades.

```
.elemento {  
    font-size: 16px;  
    width: 200px;  
}
```

REM (Root EM)

REM é uma unidade relativa que baseia seu valor no tamanho da fonte do elemento raiz (<html>). 1rem é igual ao tamanho da fonte raiz, que por padrão é 16px. Portanto, 1.5rem seria 1.5 vezes o valor da fonte raiz.

Características:

- Unidade relativa à fonte raiz.
- Escala com base no tamanho da fonte raiz.
- Facilita a criação de designs responsivos.

```
/* Supondo que o tamanho da fonte raiz seja 16px */  
.elemento {  
    font-size: 1.5rem; /* 1.5 * 16px = 24px */  
}
```

EM (Elemento)

EM é uma unidade relativa baseada no tamanho da fonte do elemento pai. Se o elemento pai tiver uma fonte de 20px, então 1em será igual a 20px. Se for 2em, será igual a duas vezes o tamanho da fonte do elemento pai (40px).

Características:

- Unidade relativa ao tamanho da fonte do elemento pai.
- Útil para criar layouts que se ajustem ao contexto.
- Pode levar a comportamentos inesperados se não for bem compreendida.

```
/* Supondo que o elemento pai tenha um tamanho de fonte de 20px */
.elemento {
    font-size: 2em; /* 2 * 20px = 40px */
}
```

Comparação entre PX, REM e EM

Unidade	Base de Cálculo	Escalável	Exemplo
PX	Valor da tela (fixo)	Não	<code>font-size: 16px</code>
REM	Tamanho da fonte raiz (<code><html></code>)	Sim	<code>font-size: 1.5rem</code> (1.5 * 16px)
EM	Tamanho da fonte do elemento pai	Sim	<code>font-size: 2em</code> (2 * 20px)

Nesta aula, vamos continuar a estilização do nosso projeto, focando nos elementos do formulário. Vamos aplicar estilos aos títulos, formulários, fieldsets, legendas, rótulos e inputs para melhorar a aparência e a usabilidade do nosso formulário.

Objetivo

- Aprender a aplicar estilos detalhados usando CSS.
- Estilizar elementos do formulário para criar uma interface mais atraente e funcional.

Passo 1: Estilizando os Títulos

Vamos começar aplicando estilos aos títulos (h1, h2, h3). Definiremos o tamanho da fonte usando a unidade relativa rem.

```
h1,  
h2,  
h3 {  
  font-size: 1.5rem;  
  /* 16px + (16/2) = 8px -> 16 + 8 = 24px */  
}
```

Explicação:

- font-size: 1.5rem: Define o tamanho da fonte como 1.5 vezes o valor da fonte raiz (16px), resultando em 24px.

Passo 2: Estilizando o Formulário

Vamos aplicar um espaço superior ao formulário para separar visualmente dos elementos acima.

```
#formulario {  
  margin-top: 15px;  
}
```

Explicação:

- margin-top: 15px: Adiciona um espaço superior de 15px ao formulário.

Passo 3: Estilizando Fieldsets

Vamos estilizar os fieldsets para adicionar uma borda, padding, bordas arredondadas e um espaçamento inferior.

```
fieldset {  
    border: 1px solid var(--input-border);  
    padding: 10px;  
    border-radius: 5px;  
    margin-bottom: 15px;  
}
```

Explicação:

- border: Define a borda com 1px de espessura e a cor da variável --input-border.
- padding: Adiciona espaço interno de 10px.
- border-radius: Define bordas arredondadas de 5px.
- margin-bottom: Adiciona um espaço inferior de 15px.

Passo 4: Estilizando Legendas

Vamos estilizar as legendas (legend) para destacar o texto usando cor de realce e aumentar o peso da fonte.

```
legend {  
    color: var(--text-realce);  
    font-weight: 700;  
    /* font-weight: bold; */  
}
```

Explicação:

- color: Define a cor do texto como a variável --text-realce.
- font-weight: Define o peso da fonte como 700 (negrito).

Passo 5: Estilizando Rótulos

Vamos aplicar estilos aos rótulos (label) para garantir que cada rótulo seja exibido em uma nova linha e adicionar espaçamento inferior.

```
label {  
  display: block;  
  margin-bottom: 5px;  
}
```

Explicação:

- display: block: Faz com que o rótulo ocupe toda a largura disponível, garantindo que cada rótulo esteja em uma nova linha.
- margin-bottom: Adiciona um espaço inferior de 5px.

Passo 6: Estilizando Inputs

Vamos estilizar os inputs do tipo texto e número para garantir uma aparência consistente e melhorar a usabilidade.

```
input[type="text"],  
input[type="number"] {  
  width: 100%;  
  padding: 10px;  
  margin-bottom: 10px;  
  border: 1px solid var(--input-border);  
  border-radius: 5px;  
  background-color: var(--input-background);  
  color: var(--white);  
  font-size: 1rem;  
}
```

Explicação:

- width: 100%: Faz com que o input ocupe toda a largura disponível.
- padding: Adiciona espaço interno de 10px.
- margin-bottom: Adiciona um espaço inferior de 10px.
- border: Define a borda com 1px de espessura e a cor da variável --input-border.
- border-radius: Define bordas arredondadas de 5px.
- background-color: Define a cor de fundo como a variável --input-background.
- color: Define a cor do texto como a variável --white.
- font-size: Define o tamanho da fonte como 1rem (16px).

Passo 7: Estilizando Inputs em Foco

Vamos aplicar estilos aos inputs quando eles estiverem em foco para melhorar a experiência do usuário.

```
input[type="text"]:focus,  
input[type="number"]:focus {  
    border-color: var(--focus);  
    outline: none;  
}
```

Explicação:

- border-color: Muda a cor da borda para a variável --focus quando o input está em foco.
- outline: none: Remove a linha de contorno padrão do navegador ao focar no input.

Conclusão

Nesta aula, aprendemos a aplicar estilizações detalhadas aos elementos do formulário usando CSS. Aplicamos estilos aos títulos, formulários, fieldsets, legendas, rótulos e inputs para criar uma interface mais atraente e funcional.

Nesta aula, vamos continuar a estilização do nosso projeto, focando nos grupos de formulário e nos botões. Vamos aplicar estilos que melhorem a aparência e a usabilidade desses elementos.

Passo 1: Estilizando Grupos de Formulário

Vamos estilizar os grupos de formulário (.form-group) para definir a disposição dos elementos dentro de cada grupo e adicionar espaçamento.

```
.form-group {  
  display: flex;  
  align-items: center;  
  gap: 10px;  
  margin-bottom: 10px;  
}
```

Explicação:

- display: flex: Utiliza o modelo flexbox para dispor os elementos do grupo.
- align-items: center: Alinha verticalmente os elementos ao centro.
- gap: 10px: Adiciona um espaçamento de 10px entre os elementos.
- margin-bottom: 10px: Adiciona um espaço inferior de 10px para separar cada grupo de formulário.

Passo 2: Estilizando o Botão do Formulário

Vamos estilizar o botão do formulário (#form_btn) para criar um botão atraente e responsivo.

```
#form_btn {  
  width: 100%;  
  padding: 10px;  
  font-size: 1em;  
  background-color: var(--button-background);  
  color: var(--white);  
  border: none;  
  border-radius: 5px;  
  cursor: pointer;  
}
```

Explicação:

- width: 100%: Faz com que o botão ocupe toda a largura disponível.
- padding: 10px: Adiciona espaço interno de 10px.
- font-size: 1em: Define o tamanho da fonte como 1em (igual ao tamanho da fonte do elemento pai).
- background-color: Aplica a cor de fundo definida na variável --button-background.
- color: var(--white): Define a cor do texto como branco.
- border: none: Remove a borda padrão do botão.
- border-radius: 5px: Define bordas arredondadas de 5px.
- cursor: pointer: Muda o cursor para uma mãozinha ao passar o mouse sobre o botão.

Estilizando o Botão ao Passar o Mouse

Vamos aplicar um estilo ao botão quando o usuário passar o mouse sobre ele (hover).

```
#form_btn:hover {  
    background-color: var(--safe);  
}
```

Explicação:

- background-color: var(--safe); Muda a cor de fundo do botão para a variável --safe ao passar o mouse sobre ele.

Passo 3: Estilizando o Botão de Desconto

Vamos estilizar o botão de desconto (#discount-btn) para criar um botão atraente e responsivo.

```
#discount-btn {  
    background-color: var(--text-realce);  
    color: var(--white);  
    padding: 10px 20px;  
    font-size: 1em;  
    border: none;  
    border-radius: 5px;  
    cursor: pointer;  
}
```

Explicação:

- background-color: Aplica a cor de fundo definida na variável --text-realce.
- color: var(--white); Define a cor do texto como branco.
- padding: 10px 20px: Adiciona espaço interno de 10px nas partes superior e inferior, e 20px nas laterais.
- font-size: 1em: Define o tamanho da fonte como 1em (igual ao tamanho da fonte do elemento pai).
- border: none: Remove a borda padrão do botão.
- border-radius: 5px: Define bordas arredondadas de 5px.
- cursor: pointer: Muda o cursor para uma mãozinha ao passar o mouse sobre o botão.

Estilizando o Botão de Desconto ao Passar o Mouse

Vamos aplicar um estilo ao botão de desconto quando o usuário passar o mouse sobre ele (hover).

```
#discount-btn:hover {  
    background-color: var(--focus);  
}
```

Explicação:

- background-color: var(--focus); Muda a cor de fundo do botão para a variável --focus ao passar o mouse sobre ele.

Nesta aula, vamos continuar a estilização do nosso projeto, focando nas seções de desconto, ícones de pagamento e garantia. Vamos aplicar estilos detalhados a essas seções para melhorar a aparência e a usabilidade da nossa página.

Passo 1: Estilizando a Seção de Desconto

Vamos começar aplicando estilos à seção de desconto para separar visualmente essa área e adicionar espaçamento.

```
.apply-discount {  
    border-top: 1px solid var(--input-border);  
    padding-top: 10px;  
    margin-top: 10px;  
}
```

Explicação:

- border-top: Adiciona uma borda superior de 1px com a cor definida na variável --input-border.
- padding-top: Adiciona um espaçamento interno superior de 10px.
- margin-top: Adiciona um espaçamento superior de 10px para separar a seção de elementos acima dela.

Vamos aplicar estilos ao cabeçalho da seção de desconto para dispor os elementos horizontalmente e melhorar a usabilidade.

```
.apply-discount-header {  
    display: flex;  
    justify-content: space-between;  
    align-items: center;  
    cursor: pointer;  
    color: var(--text-realce);  
}
```

Explicação:

- display: flex: Utiliza flexbox para dispor os elementos horizontalmente.
- justify-content: space-between: Distribui os elementos ao longo da linha com espaço igual entre eles.
- align-items: center: Alinha verticalmente os itens ao centro.
- cursor: pointer: Muda o cursor para uma mãozinha ao passar o mouse sobre o cabeçalho.
- color: Define a cor do texto como a variável --text-realce.

Vamos aplicar estilos ao corpo da seção de desconto para adicionar espaçamento superior.

```
.apply-discount-body {  
    /* display: none; */  
    margin-top: 10px;  
}
```

Explicação:

- display: none: Comentado, sugerindo que o conteúdo deve estar oculto inicialmente.
- margin-top: Adiciona um espaçamento superior de 10px.

Vamos aplicar estilos ao input da seção de desconto para ajustar sua largura e espaçamento.

```
.apply-discount-body input {  
    width: 90%;  
    margin-right: 5px;  
}
```

Explicação:

- width: Define a largura do input como 90% do contêiner pai.
- margin-right: Adiciona um espaçamento à direita de 5px.

Passo 2: Estilizando Ícones de Pagamento

Vamos estilizar a seção de ícones de pagamento para centralizar os ícones e adicionar espaçamento entre eles.

```
.icons {  
  display: flex;  
  flex-flow: row;  
  justify-content: center;  
  gap: 10px;  
  margin-bottom: 10px;  
}
```

Explicação:

- display: flex: Utiliza flexbox para dispor os ícones em linha.
- flex-flow: row: Define a direção do fluxo como linha.
- justify-content: center: Centraliza os ícones.
- gap: Adiciona um espaçamento de 10px entre os ícones.
- margin-bottom: Adiciona um espaçamento inferior de 10px.

Vamos ajustar o tamanho dos ícones para garantir que fiquem visualmente agradáveis.

```
.icons i {  
  font-size: 20px;  
}
```

Explicação:

- font-size: Define o tamanho da fonte dos ícones como 20px.



Passo 3: Estilizando a Seção de Garantia

Vamos estilizar a caixa de garantia para centralizar o conteúdo e adicionar bordas superior e inferior.

```
.garantia_box {  
    display: flex;  
    flex-flow: column;  
    align-items: center;  
    border-top: 1px solid var(--input-border);  
    border-bottom: 1px solid var(--input-border);  
}
```

Explicação:

- display: flex: Utiliza flexbox para dispor os elementos.
- flex-flow: column: Define a direção do fluxo como coluna.
- align-items: center: Centraliza os elementos.
- border-top **e** border-bottom: Adiciona bordas superior e inferior com 1px de espessura e a cor da variável --input-border.

Vamos ajustar a cor do texto para indicar segurança.

```
.safe {  
    color: var(--safe);  
}
```

Explicação:

- color: Define a cor do texto como a variável --safe.

Vamos ajustar o tamanho e as bordas da imagem para melhorar sua aparência.

```
.safe_image {  
    width: 300px;  
    border-radius: 10%;  
    margin-bottom: 10px;  
}
```

Explicação:

- width: Define a largura da imagem como 300px.
- border-radius: Define bordas arredondadas com raio de 10%.
- margin-bottom: Adiciona um espaçamento inferior de 10px.



Nesta aula, vamos focar na estilização do rodapé da nossa página. Vamos aplicar estilos aos elementos do rodapé para garantir que eles sejam visualmente agradáveis e fáceis de usar.

Passo 1: Estilizando o Elemento <footer>

Vamos começar aplicando estilos ao elemento <footer> para centralizar seu conteúdo e adicionar espaçamento superior.

```
footer {  
    text-align: center;  
    margin-top: 20px;  
}
```

Explicação:

- text-align: center: Centraliza o texto e os elementos filhos dentro do rodapé.
- margin-top: 20px: Adiciona um espaçamento superior de 20px para separar o rodapé dos elementos acima dele.

Passo 2: Estilizando Links no Rodapé

Vamos aplicar estilos aos links (<a>) dentro do rodapé para definir a cor e remover a decoração padrão do texto. Também aplicaremos um estilo diferente ao passar o mouse sobre os links (hover).

Estilo Inicial dos Links

```
footer a {  
    color: var(--text-realce);  
    text-decoration: none;  
}
```

Explicação:

- color: Define a cor dos links como a variável --text-realce.
- text-decoration: none: Remove a decoração padrão dos links (como sublinhado).

Estilo dos Links ao Passar o Mouse

```
footer a:hover {  
    text-decoration: underline;  
}
```

Explicação:

- text-decoration: underline: Adiciona um sublinhado aos links ao passar o mouse sobre eles, indicando que são clicáveis.

Passo 3: Estilizando Parágrafos no Rodapé

Vamos aplicar estilos aos parágrafos (<p>) dentro do rodapé para definir a cor do texto.

```
footer p {  
    color: var(--text);  
}
```

Explicação:

- color: Define a cor do texto nos parágrafos como a variável --text, garantindo que o texto esteja em harmonia com o resto da página.

A partir de agora, vamos começar a construir a inteligência da nossa página de formulário. Ao longo das próximas aulas, iremos implementar e explicar passo a passo o código necessário para validar os dados inseridos pelos usuários. Nesta primeira parte, vamos focar na validação dos campos de Nome, CPF e Telefone.

Função para Verificar se o Valor Contém Apenas Letras e Espaços (Validação para o Campo Nome)

Vamos começar com a função `isValidName`, que verifica se o valor inserido no campo Nome contém apenas letras e espaços.

Explicação:

- **Objetivo:** Garantir que o campo Nome contenha apenas letras e espaços.
- **Loop:** A função percorre cada caractere da string.
- **Condicional:** Verifica se o caractere é uma letra (maiúscula ou minúscula) ou um espaço.
- **Retorno:** Se encontrar um caractere inválido, retorna false. Caso contrário, retorna true.

```
function isValidName(string) {  
    // Millene - M-0 I-1 L-2... índices  
    for (let index = 0; index < string.length; index++) {  
        let char = string[index];  
  
        if (  
            !(  
                (char >= "A" && char <= "Z") ||  
                (char >= "a" && char <= "z") ||  
                char === " "  
            )  
        ) {  
            return false;  
        }  
    }  
  
    return true;  
}
```

Função para Validar o CPF (11 Dígitos Numéricos)

Agora, vamos implementar a função isValidCPF, que verifica se o valor inserido no campo CPF contém exatamente 11 dígitos numéricos.

Explicação:

- **Objetivo:** Garantir que o campo CPF contenha exatamente 11 dígitos numéricos.
- **Condição de Comprimento:** Verifica se a string tem exatamente 11 caracteres.
- **Validação Numérica:** Utiliza isNaN para garantir que todos os caracteres são numéricos.
- **Retorno:** Retorna true se ambas as condições forem satisfeitas, caso contrário, retorna false.

```
function isValidCPF(cpf) {  
    // console.log(isNaN(cpf)); // Não é numérico  
    return cpf.length === 11 && !isNaN(cpf);  
}
```

Função para Validar o Telefone (11 Dígitos Numéricos)

Por fim, vamos implementar a função `isValidPhone`, que verifica se o valor inserido no campo Telefone contém exatamente 11 dígitos numéricos.

Explicação:

- **Objetivo:** Garantir que o campo Telefone contenha exatamente 11 dígitos numéricos.
- **Condição de Comprimento:** Verifica se a string tem exatamente 11 caracteres.
- **Validação Numérica:** Utiliza `isNaN` para garantir que todos os caracteres são numéricos.
- **Retorno:** Retorna `true` se ambas as condições forem satisfeitas, caso contrário, retorna `false`.

```
function isValidPhone(phone) {  
    return phone.length === 11 && !isNaN(phone);  
}
```

Nesta aula, vamos validar os campos de CEP e Estado e integrar todas as validações em uma função principal.

Função para Validar CEP (8 Dígitos Numéricos)

Vamos começar com a função `isValidCEP`, que verifica se o valor inserido no campo CEP contém exatamente 8 dígitos numéricos.

Explicação:

- **Objetivo:** Garantir que o campo CEP contenha exatamente 8 dígitos numéricos.
- **Condição de Comprimento:** Verifica se a string tem exatamente 8 caracteres.
- **Validação Numérica:** Utiliza `isNaN` para garantir que todos os caracteres são numéricos.
- **Retorno:** Retorna `true` se ambas as condições forem satisfeitas, caso contrário, retorna `false`.

```
function isValidCEP(cep) {  
    return cep.length === 8 && !isNaN(cep);  
}
```

Função para Validar o Estado (Sigla de Dois Caracteres, Apenas Letras Maiúsculas)

Agora, vamos implementar a função `isValidState`, que verifica se o valor inserido no campo Estado é uma sigla de dois caracteres com letras maiúsculas.

Explicação:

- **Objetivo:** Garantir que o campo Estado contenha uma sigla de dois caracteres com letras maiúsculas.
- **Condição de Comprimento:** Verifica se a string tem exatamente 2 caracteres.
- **Validação de Letras:** Verifica se ambos os caracteres são letras maiúsculas (A-Z).
- **Retorno:** Retorna true se ambas as condições forem satisfeitas, caso contrário, retorna false.

```
function isValidState(state) {  
    return (  
        state.length === 2 &&  
        state[0] >= "A" &&  
        state[0] <= "Z" &&  
        state[1] >= "A" &&  
        state[1] <= "Z"  
    );  
}
```



Função Principal para Realizar Validações e Guardar Dados

Vamos implementar a função validarEGuardarFormulario, que será responsável por validar todos os campos do formulário e guardar os dados se todas as validações forem bem-sucedidas. Vamos dividir a função em partes para explicar detalhadamente cada etapa do processo.

Passo 1: Prevenir a Submissão Padrão do Formulário

A primeira coisa que precisamos fazer é impedir que o formulário seja submetido automaticamente, pois precisamos primeiro validar os dados.

```
function validarEGuardarFormulario(evento) {  
    evento.preventDefault();
```

Explicação:

- evento.preventDefault(): Esta linha impede que o formulário seja submetido antes da validação. Isso é essencial para garantir que tenhamos a chance de validar os dados primeiro.

Passo 2: Obter os Valores dos Campos de Input

Agora vamos capturar os valores inseridos pelos usuários nos campos do formulário.

```
const nome = document.getElementById("nome").value.trim();
const cpf = document.getElementById("cpf").value.trim();
const telefone = document.getElementById("telefone").value.trim();
const cep = document.getElementById("cep").value.trim();
const rua = document.getElementById("rua").value.trim();
const numero = document.getElementById("numero").value.trim();
const complemento = document.getElementById("complemento").value.trim();
const bairro = document.getElementById("bairro").value.trim();
const cidade = document.getElementById("cidade").value.trim();
const estado = document.getElementById("estado").value.trim();
```

Explicação:

- **Captura de Valores:** Utilizamos `document.getElementById("id").value.trim()` para obter o valor de cada campo de input e remover quaisquer espaços em branco no início ou no final da string usando `trim()`.

Passo 3: Verificar se Todos os Campos Estão Preenchidos

Vamos verificar se todos os campos do formulário foram preenchidos pelos usuários.

```
if (
    !nome ||
    !cpf ||
    !telefone ||
    !cep ||
    !rua ||
    !numero ||
    !complemento ||
    !bairro ||
    !cidade ||
    !estado
) {
    alert("Por favor, preencha todos os campos");
    return;
}
```

Explicação:

- **Verificação de Preenchimento:** Utilizamos uma condicional if para verificar se algum dos campos está vazio. Se qualquer campo estiver vazio, exibimos um alerta e interrompemos a execução da função usando return.

Passo 4: Validar Cada Campo Individualmente

Vamos validar cada campo individualmente utilizando as funções de validação que já implementamos. Se algum campo não passar na validação, exibimos um alerta e interrompemos a execução da função.

Validação do Nome

```
if (!isValidName(nome)) {  
    alert("O nome deve conter apenas letras e espaços");  
    return;  
}
```

Explicação:

- **Validação do Nome:** Utilizamos a função isValidName(nome) para verificar se o campo Nome contém apenas letras e espaços. Se a validação falhar, exibimos um alerta e interrompemos a função

Validação do CPF

```
if (!isValidCPF(cpf)) {  
    alert("O CPF deve conter 11 dígitos numéricos");  
    return;  
}
```

Explicação:

- **Validação do CPF:** Utilizamos a função isValidCPF(cpf) para verificar se o campo CPF contém exatamente 11 dígitos numéricos. Se a validação falhar, exibimos um alerta e interrompemos a função.

Validação do Telefone

```
if (!isValidPhone(telefone)) {  
    alert("O Telefone deve conter 11 dígitos numéricos");  
    return;  
}
```

Explicação:

- **Validação do Telefone:** Utilizamos a função isValidPhone(telefone) para verificar se o campo Telefone contém exatamente 11 dígitos numéricos. Se a validação falhar, exibimos um alerta e interrompemos a função.

Validação do CEP

```
if (!isValidCEP(cep)) {  
    alert("O CEP deve conter 8 dígitos numéricos");  
    return;  
}
```

Explicação:

- **Validação do CEP:** Utilizamos a função isValidCEP(cep) para verificar se o campo CEP contém exatamente 8 dígitos numéricos. Se a validação falhar, exibimos um alerta e interrompemos a função.

Validação do Estado

```
if (!isValidState(estado)) {  
    alert("O Estado deve ser uma sigla de letras maiúsculas");  
    return;  
}
```

Explicação:

- **Validação do Estado:** Utilizamos a função isValidState(estado) para verificar se o campo Estado contém uma sigla de duas letras maiúsculas. Se a validação falhar, exibimos um alerta e interrompemos a função.



Adicionar o Evento de Submissão do Formulário

Finalmente, vamos adicionar o evento de submissão ao formulário para chamar a função de validação quando o formulário for submetido.

```
const formulario = document.getElementById("formulario");
formulario.addEventListener("submit", validarEGuardarFormulario);
```

Explicação:

- **Selecionar Formulário:** Captura o elemento do formulário pelo ID formulario.
- **Adicionar Evento:** Adiciona um evento de submissão que chama a função validarEGuardarFormulario.

Função Principal para Realizar Validações e Guardar Dados

Vamos continuar a implementação da função validarEGuardarFormulario, agora focando na parte de armazenar os dados validados no localStorage. Esta parte da função será responsável por salvar os dados dos usuários de forma persistente e limpar o formulário após a submissão bem-sucedida.

Armazenar os Dados no localStorage

Após validar todos os campos, se todos os dados estiverem corretos, vamos armazenar as informações no localStorage. O localStorage permite que os dados sejam salvos no navegador do usuário e persistam mesmo após recarregar a página ou fechar o navegador.

Explicação:

- **Criar um Objeto com os Dados do Formulário:** Criamos um objeto chamado dadosFormulario que contém todas as informações capturadas dos campos de input.

```
const dadosFormulario = {  
    nome,  
    cpf,  
    telefone,  
    cep,  
    rua,  
    numero,  
    complemento,  
    bairro,  
    cidade,  
    estado,  
};
```



Salvar os Dados no localStorage: Utilizamos o método `setItem` do `localStorage` para salvar os dados. Convertendo o objeto `dadosFormulario` em uma string JSON usando `JSON.stringify`.

```
localStorage.setItem("dadosFormulario", JSON.stringify(dadosFormulario))
```

Limpar o Formulário: Utilizamos o método `reset` do formulário para limpar todos os campos após a submissão bem-sucedida.

```
formulario.reset();
```

Exibir uma Mensagem de Sucesso: Exibimos um alerta indicando que os dados foram salvos com sucesso.

```
alert("Dados salvos com sucesso!");
```

Nesta aula, completamos a implementação da função principal validarEGuardarFormulario, incluindo a parte de armazenamento dos dados no localStorage. Agora nossa página de formulário é capaz de validar os dados dos usuários e salvar essas informações de forma persistente.

```
// Se tudo estiver correto, salvar os dados no localStorage
const dadosFormulario = {
    nome,
    cpf,
    telefone,
    cep,
    rua,
    numero,
    complemento,
    bairro,
    cidade,
    estado,
};

localStorage.setItem("dadosFormulario", JSON.stringify(dadosFormulario));
formulario.reset();
alert("Dados salvos com sucesso!");
}
```

Na aula, vamos aprender sobre a funcionalidade de **toggle**, que significa alternar entre dois estados possíveis. É frequentemente usado em interfaces de usuário e programação para representar algo que pode ser "ligado/desligado" ou "ativo/inativo". Vamos implementar uma função que alterna a exibição do formulário de cupom de desconto.

Função para Alternar a Exibição do Formulário de Cupom

Vamos começar implementando a função `toggleDiscount`, que será responsável por alternar a visibilidade do formulário de cupom de desconto e mudar o ícone de seta correspondente.

```
function toggleDiscount() {
  const discountBody = document.getElementById("apply-discount-body");
  const icon = document.getElementById("toggle-icon");

  // Alterna a visibilidade do formulário de desconto
  if (
    discountBody.style.display === "none" ||
    discountBody.style.display === ""
  ) {
    discountBody.style.display = "block";
    icon.classList.add("bx-chevron-up");
    icon.classList.remove("bx-chevron-down");
  } else {
    discountBody.style.display = "none";
    icon.classList.add("bx-chevron-down");
    icon.classList.remove("bx-chevron-up");
  }
}
```

Obter Elementos: Capturamos os elementos apply-discount-body e toggle-icon pelo ID.

```
const discountBody = document.getElementById("apply-discount-body");
const icon = document.getElementById("toggle-icon");
```

Verificar e Alternar a Visibilidade: Verificamos a visibilidade atual do formulário de desconto e alternamos entre "block" e "none"

```
if (
  discountBody.style.display === "none" ||
  discountBody.style.display === ""
) {
  discountBody.style.display = "block";
} else {
  discountBody.style.display = "none";
}
```

Mudar o Ícone: Alteramos a classe do ícone para refletir a mudança de estado (seta para cima ou para baixo).

```
if (
  discountBody.style.display === "none" ||
  discountBody.style.display === ""
) {
  icon.classList.add("bx-chevron-up");
  icon.classList.remove("bx-chevron-down");
} else {
  icon.classList.add("bx-chevron-down");
  icon.classList.remove("bx-chevron-up");
}
```



Adicionando o Toggle na Interface

Para que a função toggleDiscount funcione corretamente, precisamos garantir que o HTML contenha os elementos necessários e que a função seja chamada no evento de clique.

```
<div class="apply-discount">
  <div class="apply-discount-header" onclick="toggleDiscount()">
    <h3>Aplicar Cupom de Desconto</h3>
    <i id="toggle-icon" class="bx bx-chevron-down"></i>
  </div>
  <div id="apply-discount-body" style="display: none;">
    <input type="text" placeholder="Digite seu cupom" />
    <button id="discount-btn">Aplicar</button>
  </div>
</div>
```

Explicação:

- apply-discount-header: Contém o cabeçalho do formulário de desconto e chama a função toggleDiscount no evento de clique.
- toggle-icon: Ícone que muda entre seta para cima (bx-chevron-up) e seta para baixo (bx-chevron-down) ao alternar a visibilidade.
- apply-discount-body: Contém o corpo do formulário de desconto e é inicialmente oculto (style="display: none;").

Vamos criar uma função que verifica se o cupom de desconto inserido é válido, aplica o desconto ao preço total, exibe a mensagem correspondente e armazena o cupom no localStorage.

Estrutura de Cupons de Desconto

Primeiro, vamos definir uma estrutura para armazenar os cupons de desconto e seus valores correspondentes.

```
const discountCupons = {  
    DESCONT010: 0.1, // 10% DESCONTO  
    DESCONT020: 0.2, // 20% DESCONTO  
    DESCONT050: 0.5, // 50% DESCONTO  
};
```

Explicação:

- **Objeto** discountCupons: Contém pares chave-valor onde as chaves são os códigos dos cupons e os valores são as porcentagens de desconto correspondentes.

Função para Aplicar o Desconto

Vamos implementar a função `applyDiscount`, que será responsável por aplicar o desconto ao preço total.

Obter o Valor do Cupom: Capturamos o valor inserido pelo usuário no campo de input e o convertemos para letras maiúsculas.

```
const discountCode = document
  .getElementById("discount-code")
  .value.trim()
  .toUpperCase();
```

Obter Elementos de Mensagem e Preço: Capturamos os elementos onde a mensagem de desconto e o preço total serão exibidos.

```
const discountMessageElement = document.getElementById("discount-message");
const totalPriceElement = document.getElementById("total-price");
```

Verificar Validade do Cupom: Verificamos se o código de desconto inserido pelo usuário existe no objeto discountCupons.

```
if (discountCupons[discountCode]) {
```

Calcular o Preço com Desconto: Calculamos o preço com desconto aplicando a porcentagem correspondente ao preço original.

```
const discount = discountCupons[discountCode];
const originalPrice = 1200;
const discountedPrice = originalPrice * (1 - discount);
```

Atualizar o Preço Total: Atualizamos o elemento de preço total com o valor descontado.

```
//Atualizar o preço com o desconto
totalPriceElement.innerText = `Preço Total: R$ ${discountedPrice.toFixed(
  2
)}`;
```

Armazenar o Cupom no localStorage: Salvamos o código do desconto no localStorage para futura referência.

```
//Exibir a mensagem de desconto
discountMessageElement.style.color = "green";
discountMessageElement.innerText = `Desconto de ${discountCode} aplicado!`;
```

Exibir Mensagem de Sucesso: Exibimos uma mensagem verde indicando que o desconto foi aplicado com sucesso.

```
localStorage.setItem("discount", discountCode);
```

Limpar Campo de Input: Limpamos o campo de input de desconto após aplicar o cupom.

```
document.getElementById("discount-code").value = "";
```

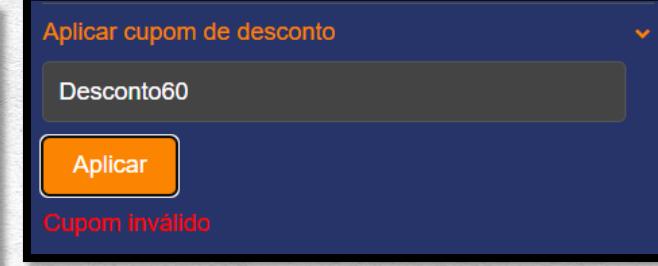
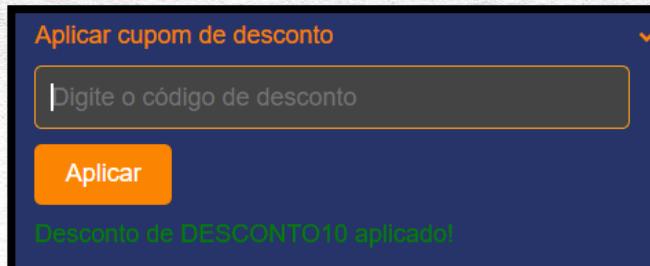
Mensagem de Cupom Inválido: Se o código de desconto não for válido, exibimos uma mensagem vermelha de erro.

```
} else {  
    discountMessageElement.style.color = "red";  
    discountMessageElement.innerText = "Cupom inválido";  
}
```

Adicionando a Função applyDiscount ao Botão

Para que a função applyDiscount funcione corretamente, precisamos garantir que o botão de aplicar desconto chame essa função no evento de clique.

```
<div class="apply-discount-body" id="apply-discount-body">  
    <input type="text" id="discount-code" placeholder="Digite o código de desconto" />  
    <button id="discount-btn" onclick="applyDiscount()>Aplicar</button>  
    <span id="discount-message"></span>  
</div>  
<p id="total-price">Preço Total: R$ 1200.00</p>
```



Vamos aprender a implementar uma função que verifica se há um cupom de desconto armazenado no localStorage quando a página é carregada. Se houver, a função irá limpar o cupom armazenado e atualizar a mensagem de desconto. Vamos explicar passo a passo como implementar e usar essa função.

Função para Verificar e Limpar Cupom de Desconto Armazenado

Vamos implementar a função checkStoredDiscount, que será responsável por verificar se há um cupom de desconto armazenado no localStorage e, se houver, limpar o desconto e atualizar a mensagem de desconto.

```
function checkStoredDiscount() {
    const storedDiscount = localStorage.getItem("discount");
    const discountMessageElement = document.getElementById("discount-message");

    //Verificar se há cupom no localStorage
    if (storedDiscount) {
        //limpar o desconto
        localStorage.removeItem('discount');
    }

    discountMessageElement.innerText = "";
}

window.onload = checkStoredDiscount;
```

Explicação:

- **Obter o Cupom Armazenado:** Capturamos o valor do cupom armazenado no localStorage usando localStorage.getItem("discount").

```
const storedDiscount = localStorage.getItem("discount");
```

- **Obter o Elemento de Mensagem:** Capturamos o elemento onde a mensagem de desconto será exibida.

```
const discountMessageElement = document.getElementById("discount-message");
```

- **Verificar e Limpar o Cupom:** Verificamos se há um cupom armazenado. Se houver, removemos o cupom do localStorage.

```
if (storedDiscount) {  
    localStorage.removeItem("discount");  
}
```

Atualizar a Mensagem de Desconto: Limpamos qualquer mensagem de desconto exibida anteriormente.

```
discountMessageElement.innerText = "";
```

Executar a Função ao Carregar a Página

Para garantir que a função checkStoredDiscount seja executada automaticamente quando a página for carregada, vamos atribuí-la ao evento window.onload.

```
window.onload = checkStoredDiscount;
```

Explicação:

- **Evento** window.onload: Garantimos que a função checkStoredDiscount seja chamada automaticamente quando a página for carregada, verificando e limpando qualquer cupom de desconto armazenado.

Módulo 11

INTRODUÇÃO AO BACK-END

INTRODUÇÃO AO BACK-END

INTRODUÇÃO AO BACK-END



Definição de Backend: O Backend é a parte de um sistema de software responsável pelo processamento dos dados e pela lógica de negócios. Enquanto o Frontend lida com a interface visível para o usuário, o Backend trabalha nos bastidores para garantir que tudo funcione corretamente.

Papel do Backend no desenvolvimento Web

O Backend desempenha várias funções essenciais no desenvolvimento web:

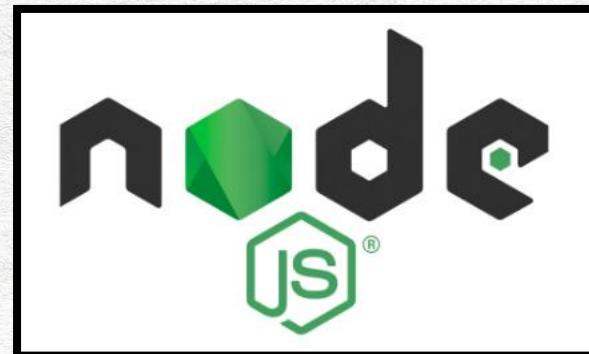
- **Processamento de Dados:** É responsável por receber os dados do Frontend, processá-los conforme necessário e gerar respostas apropriadas para o cliente.
- **Acesso a Banco de Dados:** O Backend interage com sistemas de gerenciamento de banco de dados para armazenar, recuperar, atualizar e excluir informações conforme solicitado pela aplicação.
- **Lógica de Negócios:** Implementa as regras e operações específicas do negócio que governam o comportamento da aplicação, garantindo que as operações sejam realizadas de maneira consistente e segura.
- **Segurança:** Implementa medidas de segurança para proteger os dados e garantir que apenas usuários autorizados possam acessar informações sensíveis.
- **Integração de Sistemas:** Permite a integração com serviços externos, APIs e outras aplicações para fornecer funcionalidades adicionais e melhorar a interoperabilidade do sistema.

Essas funções trabalham em conjunto para garantir o bom funcionamento e a segurança das aplicações web, tornando o Backend uma parte crucial do desenvolvimento de software.

No vasto panorama do desenvolvimento web, o Backend desempenha um papel crucial na construção de sistemas robustos e eficientes. Entre as tecnologias comumente empregadas nesse domínio, destacam-se Python, Ruby on Rails, PHP e Node.js. No entanto, é o último, Node.js, que merece uma atenção especial. Node.js não é apenas uma tecnologia, mas sim um ambiente de tempo de execução JavaScript que revolucionou a forma como o código do lado do servidor é concebido e implementado.

Permitindo aos desenvolvedores executar JavaScript não apenas no navegador, mas também no servidor, o Node.js abre as portas para uma abordagem unificada e coesa no desenvolvimento de aplicações web. Neste texto, exploraremos mais profundamente o papel e as capacidades do Node.js no mundo do desenvolvimento backend.

Além disso, o Node.js é conhecido por sua arquitetura orientada a eventos e assíncrona, o que o torna altamente eficiente e escalável para lidar com grandes volumes de conexões simultâneas. Essa abordagem conceitual permite que as aplicações desenvolvidas com Node.js sejam extremamente responsivas e capazes de lidar com requisitos de alto desempenho.



Processamento de Dados:

- O que é: Validar e processar informações enviadas por formulários, realizar cálculos complexos, e formatar dados para apresentação.
- Por que é importante: É essencial garantir que os dados enviados pelos usuários sejam corretos e úteis para a aplicação. Além disso, o processamento adequado dos dados é fundamental para fornecer uma experiência de usuário eficiente e precisa.
- Como fazer: Utilizando funções e métodos para validar entradas, realizar operações matemáticas e formatar saídas conforme necessário.

Acesso a Banco de Dados:

- O que é: Consultar, inserir, atualizar e excluir dados em um banco de dados, permitindo que a aplicação armazene e recupere informações de forma organizada.
- Por que é importante: Bancos de dados são essenciais para armazenar e gerenciar grandes volumes de dados de forma eficiente, garantindo a integridade e a disponibilidade das informações.
- Como fazer: Utilizando bibliotecas e frameworks para se conectar e interagir com o banco de dados, escrevendo consultas SQL para recuperar e manipular os dados conforme necessário.

Autenticação de Usuário:

- O que é: Gerenciar o processo de login e logout de usuários na aplicação, verificando suas credenciais e fornecendo acesso seguro aos recursos protegidos.
- Por que é importante: A autenticação é crucial para proteger as informações dos usuários e garantir que apenas pessoas autorizadas tenham acesso aos recursos da aplicação.
- Como fazer: Implementando sistemas de login seguro, utilizando técnicas como hashing de senhas, tokens JWT (JSON Web Tokens) para autenticação de API e cookies para manter a sessão do usuário.

Autorização de Acesso:

- *O que é:* Controlar e gerenciar as permissões de acesso dos usuários a diferentes partes e funcionalidades da aplicação.
- *Por que é importante:* Garantir que os usuários tenham acesso apenas ao que é necessário para realizar suas tarefas, protegendo assim os dados sensíveis e mantendo a segurança da aplicação.
- *Como fazer:* Implementando sistemas de controle de acesso baseados em funções (RBAC), definindo diferentes níveis de permissão para usuários e grupos de usuários.

Envio de Emails:

- *O que é:* Enviar mensagens de email para os usuários, como confirmações de cadastro, redefinições de senha e notificações de atividade.
- *Por que é importante:* Email é uma forma eficaz de se comunicar com os usuários e fornecer informações importantes sobre suas contas e atividades na aplicação.
- *Como fazer:* Utilizando bibliotecas de email para enviar mensagens programaticamente, integrando serviços de email como SMTP ou APIs de provedores de email.

Processamento de Arquivos:

- *O que é:* Manipular arquivos enviados pelos usuários, como uploads de imagens e documentos, armazenando e recuperando esses arquivos de forma eficiente.
- *Por que é importante:* Permitir que os usuários enviem e compartilhem arquivos é uma funcionalidade comum em muitas aplicações, como redes sociais e plataformas de compartilhamento de arquivos.
- *Como fazer:* Utilizando bibliotecas para lidar com o armazenamento e manipulação de arquivos, implementando sistemas de gerenciamento de arquivos para organizar e recuperar os dados conforme necessário.

Agendamento de Tarefas:

- *O que é:* Programar a execução de tarefas em segundo plano, como processamento de filas, geração de relatórios e limpeza de dados obsoletos.
- *Por que é importante:* Automatizar tarefas repetitivas e demoradas pode melhorar significativamente a eficiência e o desempenho da aplicação, liberando recursos para outras atividades.
- *Como fazer:* Utilizando bibliotecas e ferramentas de agendamento de tarefas para executar processos em segundo plano de forma confiável e escalável.

Nesta aula vamos conhecer sobre uma das principais funcionalidades do backend, que é sua capacidade de criar um servidos, e para entendermos esses conceito vamos utilizar o exemplo abaixo:

```
const http = require("http"); // Importa o módulo HTTP do Node.js, que fornece funcionalidades para criar servidores web.

const server = http.createServer((req, res) => { // Cria um servidor utilizando o método createServer do módulo HTTP.
    // Esta função é chamada toda vez que o servidor recebe uma requisição HTTP.

    res.writeHead(200, {"Content-type": "text/html"}); // Escreve o cabeçalho da resposta HTTP.
    // O código 200 indica que a requisição foi bem-sucedida.
    // O segundo argumento é um objeto que define os cabeçalhos da resposta. Aqui, estamos especificando que o conteúdo será HTML.

    res.write("<h1>Olá, mundo!</h1>"); // Escreve o corpo da resposta HTTP.
    // Neste caso, estamos enviando uma mensagem HTML que será exibida no navegador.

    res.end(); // Finaliza a resposta HTTP, indicando ao cliente que todas as informações foram enviadas.
});

server.listen(3000, () => { // Inicia o servidor e o faz escutar as requisições na porta 3000.
    console.log("Servidor rodando na porta 3000...."); // Exibe uma mensagem no console quando o servidor estiver pronto para receber requisições.
});
```

- **Módulo HTTP:** No Node.js, os módulos são arquivos ou bibliotecas que contêm funções e variáveis para realizar tarefas específicas. O módulo HTTP é utilizado para lidar com a criação de servidores web e o processamento de requisições HTTP.
- **createServer():** Este método cria um servidor HTTP que irá escutar por requisições HTTP. Ele recebe uma função de callback que será executada toda vez que o servidor receber uma requisição.
- **Requisição (req) e Resposta (res):** São objetos que representam a requisição feita pelo cliente ao servidor e a resposta enviada pelo servidor ao cliente, respectivamente. Através destes objetos, é possível acessar informações sobre a requisição (como URL, cabeçalhos e corpo) e enviar uma resposta de volta ao cliente.
- **writeHead():** Este método define o cabeçalho da resposta HTTP que será enviada ao cliente. Aqui, estamos especificando o código de status da resposta (200 para indicar sucesso) e o tipo de conteúdo que será enviado (no caso, HTML).
- **write():** Este método é utilizado para escrever o corpo da resposta HTTP. Aqui, estamos enviando uma mensagem HTML que será exibida no navegador do cliente.
- **end():** Este método finaliza a resposta HTTP, indicando ao cliente que todas as informações foram enviadas. Após chamar este método, não é possível adicionar mais conteúdo à resposta.
- **listen():** Este método inicia o servidor e o faz escutar por requisições na porta especificada (no caso, a porta 3000). Quando o servidor estiver pronto para receber requisições, a função de callback é executada.
- **Console.log():** Esta função é utilizada para exibir mensagens no console do Node.js. Aqui, estamos exibindo uma mensagem indicando que o servidor está rodando e pronto para receber requisições.

Linhas de Comando (CLI)

As Linhas de Comando Interface de Linha de Comando (CLI) são uma forma de interagir com um computador ou sistema operacional usando texto simples. Em vez de clicar em ícones ou menus gráficos, você digita comandos específicos em uma janela de terminal ou prompt de comando e pressiona "Enter" para executá-los.

Esses comandos podem variar de tarefas simples, como criar um novo diretório ou copiar arquivos, a operações mais avançadas, como configurar redes, gerenciar processos ou automatizar tarefas complexas.

A CLI é frequentemente usada em ambientes de desenvolvimento, administração de sistemas e operações de rede, devido à sua eficiência e flexibilidade. Ela permite que os usuários realizem tarefas rapidamente, especialmente quando precisam lidar com muitos arquivos ou realizar operações repetitivas.

Embora possa ter uma curva de aprendizado inicial, dominar a CLI pode ser incrivelmente poderoso para usuários avançados, oferecendo controle total sobre o sistema e permitindo a automação de tarefas, o que pode economizar tempo e minimizar erros.

```
PS C:\Users\mikag\OneDrive\Documentos\AulasJavascript\Backend\Aula 5> node ./script.js
Utilizando linha de comando
```

As Linhas de Comando Interface de Linha de Comando (CLI) servem para uma variedade de propósitos, incluindo:

- **Automatização de Tarefas:** A CLI permite a criação de scripts que automatizam tarefas repetitivas. Isso é útil em administração de sistemas, desenvolvimento de software e operações de rede, onde muitas vezes as mesmas tarefas precisam ser executadas várias vezes.
- **Gerenciamento de Arquivos e Diretórios:** Com a CLI, você pode criar, copiar, mover, renomear e excluir arquivos e diretórios de forma eficiente.
- **Administração de Sistema:** A CLI oferece acesso rápido a configurações e recursos do sistema operacional, permitindo a instalação de software, gerenciamento de usuários, monitoramento de recursos do sistema e diagnóstico de problemas.
- **Desenvolvimento de Software:** Muitas ferramentas de desenvolvimento, como compiladores, depuradores e sistemas de controle de versão, são acessadas via CLI. Isso permite que os desenvolvedores automatizem tarefas de compilação, teste e implantação.
- **Gerenciamento de Rede:** Com a CLI, você pode configurar e monitorar dispositivos de rede, como roteadores, switches e firewalls. Isso inclui a configuração de endereços IP, rotas, VLANs e políticas de segurança.
- **Análise e Manipulação de Dados:** A CLI oferece poderosas ferramentas para análise e manipulação de dados, incluindo filtragem, ordenação, agrupamento e transformação de dados em formatos como texto, CSV e JSON.
- **Segurança:** A CLI pode ser usada para realizar tarefas relacionadas à segurança, como criptografia de arquivos, verificação de integridade de arquivos, análise de logs e monitoramento de atividades suspeitas.

O comando **cd** é uma abreviação de "Change Directory" (Mudar Diretório). Ele é usado para navegar entre diretórios em um sistema de arquivos. Aqui está como funciona:

- 1. cd**: Quando você digita apenas **cd** seguido ou não de um caminho, você volta para o diretório inicial do usuário. Por exemplo, se você estiver em **/home/usuario/documentos** e digitar **cd**, você será direcionado para o diretório inicial do usuário, geralmente em **/home/usuario**.
- 2. cd [caminho]**: Você pode usar o comando **cd** seguido do nome do diretório para o qual deseja navegar. Por exemplo, se você estiver em **/home/usuario** e quiser ir para o diretório **documentos**, você digita **cd documentos**.
- 3. cd ..**: Este é um caso especial do comando **cd**. O **..** significa "diretório pai", então quando você digita **cd ..**, você sobe um nível na hierarquia de diretórios. Por exemplo, se você estiver em **/home/usuario/documentos** e digitar **cd ..**, você será direcionado para **/home/usuario**.
4. Esses são os comandos básicos do **cd**. Eles são amplamente utilizados para navegar pelo sistema de arquivos em sistemas operacionais baseados em Unix, como Linux e macOS, assim como no prompt de comando do Windows.

O comando **ls** é usado para listar os arquivos e diretórios em um diretório específico. Aqui estão algumas maneiras comuns de usar o comando **ls**:

- **ls**: Quando você digita apenas **ls**, ele lista os arquivos e diretórios no diretório atual em que você está.
- **ls [caminho]**: Você pode especificar um caminho para listar os arquivos e diretórios em um diretório diferente. Por exemplo, **ls /home/usuario/documentos** lista os arquivos e diretórios dentro do diretório "documentos".
- **ls -l**: O uso da opção **-l** (minúsculo L) exibe uma lista detalhada que inclui informações como permissões, proprietário, grupo, tamanho, data de modificação e nome do arquivo.
- **ls -a**: A opção **-a** lista todos os arquivos, incluindo os arquivos ocultos. Arquivos ocultos são aqueles cujos nomes começam com um ponto (.) no Unix. Eles são ocultos por padrão para evitar que apareçam em listagens normais.
- **ls -h**: A opção **-h** torna os tamanhos de arquivo humanamente legíveis. Por exemplo, em vez de mostrar o tamanho do arquivo em bytes, ele usa unidades como KB, MB, GB, etc.
- **ls -F**: A opção **-F** exibe um caractere especial ao lado de cada nome de arquivo para indicar o tipo de arquivo. Por exemplo, uma barra (/) indica um diretório, um asterisco (*) indica um arquivo executável e um arroba (@) pode indicar um link simbólico.

```
Diretório: C:\Users\mikag\OneDrive\Documentos\AulasJavascript\Backend

Mode LastWriteTime Length Name
---- ----- ----- -----
da---l 20/03/2024   18:49      Aula 4
da---l 20/03/2024   18:50      Aula 5
```

O npm (Node Package Manager) é uma ferramenta essencial para desenvolvedores JavaScript que desejam gerenciar bibliotecas e pacotes de código em seus projetos. Ele é especialmente útil quando você está trabalhando com o Node.js, um ambiente de tempo de execução JavaScript que permite que você execute JavaScript no lado do servidor.

O que é o npm?

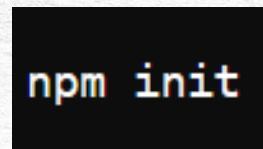
O npm é um gerenciador de pacotes que vem integrado com o Node.js. Ele permite que você instale, compartilhe e gerencie pacotes de código JavaScript facilmente. Um pacote pode ser uma biblioteca útil, um framework ou até mesmo uma aplicação pronta para uso.

Por que o npm é útil?

- **Gerenciamento de Dependências:** Com o npm, você pode facilmente instalar e atualizar as dependências do seu projeto. Isso é crucial para garantir que seu projeto tenha acesso às últimas funcionalidades e correções de bugs.
- **Reutilização de Código:** O npm permite que você use facilmente código criado por outros desenvolvedores. Você pode explorar uma vasta biblioteca de pacotes disponíveis no registro do npm e integrá-los em seu projeto com apenas alguns comandos.
- **Facilidade de Colaboração:** Com o npm, é fácil colaborar em projetos de código aberto. Você pode compartilhar seu código com outros desenvolvedores ou contribuir para projetos existentes, tornando a colaboração mais eficiente.

Como começar com o npm?

Para começar a usar o npm em seu projeto, você primeiro precisa inicializar um arquivo **package.json**. O **package.json** é um arquivo de manifesto que descreve seu projeto e suas dependências. Você pode criar este arquivo executando o comando **npm init** no terminal.



Ao executar o comando **npm init**, você será solicitado a fornecer algumas informações sobre seu projeto, como o nome do projeto, a versão, a descrição, o ponto de entrada do script e muito mais. Essas informações comporão o arquivo **package.json** e serão úteis para outros desenvolvedores que queiram usar ou colaborar com seu projeto.

```
Press ^C at any time to quit.  
package name: (aula-6)  
version: (1.0.0)  
description: Primeiro npm init  
entry point: (script.js)  
test command:  
git repository:  
keywords:  
author: Millene  
license: (ISC)
```

```
{  
  "name": "aula-6",  
  "version": "1.0.0",  
  "description": "Primeiro npm init",  
  "main": "script.js",  
  "scripts": {  
    "test": "echo \\"$Error: no test specified\\\" && exit 1"  
  },  
  "author": "Millene",  
  "license": "ISC"  
}  
  
Is this OK? (yes)
```



Uma flag na linha de comando é uma opção que você pode fornecer junto com um comando para modificar ou personalizar o comportamento desse comando. As flags são frequentemente representadas por um ou mais caracteres precedidos por um hífen (-) ou dois hifens (--). Elas são usadas para passar informações adicionais para o comando que está sendo executado.

No contexto do npm, o comando **npm init** é usado para inicializar um novo projeto npm, criando um arquivo **package.json** que contém informações sobre o projeto, como nome, versão, descrição, dependências e outras configurações.

A flag **-y** (ou **--yes**) no comando **npm init -y** é uma abreviação para "sim" (yes). Quando você usa essa flag, o npm irá automaticamente aceitar todas as opções padrão ao criar o arquivo **package.json**, sem solicitar a confirmação do usuário para cada uma delas. Isso é útil quando você deseja inicializar rapidamente um novo projeto sem ter que responder a uma série de perguntas interativas.



The screenshot shows a terminal window with a dark theme. In the top left, there are two tabs: one labeled 'package.json' and another labeled 'script.js'. The main area of the terminal displays the following JSON code:

```
1  {
2    "name": "aula-6",
3    "version": "1.0.0",
4    "description": "",
5    "main": "script.js",
6    "scripts": {
7      "test": "echo \\\"Error: no test specified\\\" && exit 1"
8    },
9    "keywords": [],
10   "author": "",
11   "license": "ISC"
12 }
```

Dentro do arquivo **package.json**, temos uma seção chamada "**scripts**". Esta seção é usada para definir comandos que podem ser executados usando o **npm run**.

```
"scripts": {  
  "test": "echo \"Error: no test specified\" && exit 1"  
},
```

Aqui está o que essa linha de código faz:

- "**testnpm**, o script chamado **test** é frequentemente usado para executar testes automatizados. No entanto, neste caso específico, o script **test** está sendo usado para imprimir uma mensagem de erro indicando que nenhum teste foi especificado.
- "**echo "Error: no test specified"echo** é usado para imprimir uma mensagem no terminal. Neste caso, estamos imprimindo a mensagem "Error: no test specified". As aspas duplas em torno da mensagem são escapadas (\") para garantir que a mensagem seja interpretada corretamente.
- **&&**: Este é um operador de controle de fluxo em shell. Ele permite encadear comandos, garantindo que o próximo comando só seja executado se o comando anterior for bem-sucedido. Neste caso, se o comando **echo** for bem-sucedido (o que sempre é), o próximo comando, **exit 1**, será executado.
- "**exit 1exit** é usado para sair do script com um código de status específico. Neste caso, estamos saindo com o código de status **1**, que indica um erro. Isso é comum em scripts de teste, onde um código de status diferente de zero é usado para indicar que um teste falhou.

É possível criar novos comandos em **scripts** dentro do **package.json**. Você pode adicionar quantos scripts personalizados quiser para automatizar tarefas de desenvolvimento, como iniciar o servidor local, construir o projeto, executar tarefas de linting, entre outros. Esses scripts podem então ser executados usando o comando **npm run nome_do_script**.

```
npm run test
```

- **run** é um subcomando do **npm** que permite executar scripts definidos no arquivo **package.json**. Ele é usado para iniciar scripts específicos que foram configurados no projeto.

Criando Novos Comandos com **npm run**:

Além do script **test**, você pode definir outros scripts personalizados no arquivo **package.json**.

Esses scripts podem ser usados para automatizar tarefas comuns de desenvolvimento, como construir o projeto, iniciar o servidor local, executar tarefas de linting, etc.

Para adicionar um novo script, basta adicionar uma nova entrada à seção **scripts** do arquivo **package.json**.

```
"scripts": {  
  "test": "echo \"Error: no test specified\" && exit 1",  
  "script": "node ./script.js"  
},
```

```
PS C:\Users\mikag\OneDrive\Documentos\AulasJavascript\Backend\Aula 6> npm run script  
> aula-6@1.0.0 script  
> node ./script.js  
  
Utilizando CLI - npm
```



O comando **npm install** é usado para instalar pacotes JavaScript e suas dependências em um projeto Node.js. Aqui está uma explicação detalhada de como funciona e o que ele faz:

- **Instalação de Pacotes:** O principal objetivo do comando **npm install** é instalar pacotes JavaScript. Esses pacotes podem ser bibliotecas de terceiros, frameworks, utilitários ou qualquer outro tipo de código JavaScript que você deseja usar em seu projeto.
- **Dependências:** Quando você instala um pacote usando **npm install**, o npm também instala todas as dependências necessárias para que esse pacote funcione corretamente. Isso significa que, se um pacote depende de outros pacotes para funcionar, o npm cuidará de instalar essas dependências automaticamente.
- **Arquivo package.json:** O npm mantém um registro de todos os pacotes instalados em um projeto no arquivo **package.json**. Este arquivo contém informações sobre o nome do pacote, a versão, bem como as dependências necessárias para o projeto. Quando você executa **npm install**, o npm verifica o arquivo **package.json** para determinar quais pacotes precisam ser instalados e quais versões são compatíveis com o projeto.
- **Opções de Instalação:** O comando **npm install** suporta várias opções que permitem personalizar a instalação de pacotes. Por exemplo, você pode instalar um pacote globalmente usando a opção **-g**, instalar um pacote específico para o ambiente de desenvolvimento usando a opção **--save-dev**, ou instalar uma versão específica de um pacote usando **@**.
- **Cache de Pacotes:** O npm mantém um cache local de pacotes baixados para evitar baixar o mesmo pacote repetidamente. Isso ajuda a acelerar o processo de instalação, especialmente em projetos grandes com muitas dependências.

The screenshot shows a code editor with a dark theme. The file being edited is `package.json`. The `dependencies` section contains the following code:

```
{ "name": "aula-6", "version": "1.0.0", "description": "", "main": "script.js", "scripts": { "test": "echo \\\"Error: no test specified\\\" && exit 1", "script": "node ./script.js" }, "keywords": [], "author": "", "license": "ISC", "dependencies": { "jest": "^29.7.0" } }
```

Below the code editor, there is a navigation bar with tabs: PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL, and PORTS. The TERMINAL tab is selected. In the terminal window, the command `npm install jest` is run, and the output is displayed:

```
PS C:\Users\mikag\OneDrive\Documentos\AulasJavascript\Backend\Aula 6> npm install jest
added 283 packages, and audited 284 packages in 2m
32 packages are looking for funding
  run `npm fund` for details
```

Suponha que você queira instalar o Jest, que é um framework de teste para JavaScript. Para fazer isso, você pode executar o seguinte comando no terminal:

```
npm install jest
```

Este comando instrui o npm a instalar o pacote Jest em seu projeto. O npm irá então baixar o pacote Jest e suas dependências, se houver, e instalá-los localmente no diretório do seu projeto. O progresso da instalação será exibido no terminal, e uma vez concluído, você terá acesso ao Jest em seu projeto.

- **npm run start:**

- Este comando é usado para executar o script chamado "start" definido no arquivo **package.json** na seção "**scripts**".
- Quando você executa **npm run start**, o npm procura pelo script "start" dentro do seu arquivo **package.json** e executa o comando associado a ele.
- Isso é útil quando você precisa executar um script específico que foi definido em **package.json**.

- **npm start:**

- Este comando é um atalho comumente usado para **npm run start**.
- Quando você executa **npm start**, o npm automaticamente procura pelo script chamado "start" no **package.json** e executa o comando associado a ele.
- É uma forma mais curta e conveniente de executar o script "start" sem precisar especificar explicitamente **run**.

Em resumo, **npm run start** e **npm start** são essencialmente os mesmos em termos de funcionalidade, mas **npm start** é mais comumente usado por ser mais curto e conveniente. Ambos são usados para iniciar um projeto Node.js, geralmente para iniciar um servidor local ou aplicação.

```
"scripts": {  
  "test": "echo \"Error: no test specified\" && exit 1",  
  "script": "node ./script.js",  
  "start": "jest"  
},
```

```
PS C:\Users\mikag\OneDrive\Documentos\AulasJavascript\Backend\Aula 6> npm run start  
> aula-6@1.0.0 start  
> jest
```

```
PS C:\Users\mikag\OneDrive\Documentos\AulasJavascript\Backend\Aula 6> npm start  
> aula-6@1.0.0 start  
> node ./script.js
```

API (Interface de Programação de Aplicações)

- Uma API, ou Interface de Programação de Aplicações, é um conjunto de regras e definições que permite que diferentes programas de computador se comuniquem entre si. Em termos simples, é um intermediário que estabelece como os sistemas podem interagir, compartilhando dados e funcionalidades.

Web API: Ampliando o Alcance pela Internet

- Uma Web API é uma extensão da API que utiliza a internet como seu meio de comunicação principal. Ela permite que aplicativos e websites acessem recursos online e interajam com serviços remotos, utilizando solicitações HTTP para transferir dados pela web.

Solicitações HTTP: A Linguagem da Comunicação Web

- As solicitações HTTP, baseadas no Protocolo de Transferência de Hipertexto, são utilizadas para transferir dados na web. Elas permitem que clientes, como navegadores da web, solicitem informações de servidores web e enviem dados para eles. As Web APIs utilizam essas solicitações para realizar operações dinâmicas e obter dados online.

API (Interface de Programação de Aplicações):

- Uma API é um conjunto de regras e definições que permite que diferentes programas de computador se comuniquem entre si.
- Ela é utilizada para facilitar a integração e a interação entre sistemas diversos, permitindo o compartilhamento de dados e funcionalidades.

Web API:

- Uma Web API é uma extensão da API que utiliza a internet como seu meio de comunicação principal.
- Ela é projetada especificamente para ser acessada e utilizada através da web, permitindo que aplicativos e websites interajam com recursos online.
- A principal diferença é o contexto de utilização: enquanto uma API pode ser utilizada em diferentes contextos, uma Web API é especificamente adaptada para operar na web.

Em resumo, todas as Web APIs são APIs, mas nem todas as APIs são necessariamente Web APIs. Enquanto uma API pode ser utilizada para comunicação entre diferentes sistemas, uma Web API é direcionada para a interação com recursos online, utilizando os protocolos da web, como HTTP, como meio de comunicação.

Explorando o Potencial do JSON em Interações de Aplicações

JSON (JavaScript Object Notation) é uma ferramenta indispensável na comunicação e intercâmbio de dados entre sistemas e aplicações. Sua estruturação simples e flexível o torna um formato de escolha para representar informações em uma ampla gama de cenários, desde a transmissão de dados em aplicativos web até a integração de serviços em APIs.

Entendendo os Objetos JSON

Em sua essência, um objeto JSON é uma coleção de pares de chave e valor, organizados de forma a representar dados de maneira hierárquica e estruturada. Cada par de chave e valor define uma propriedade específica do objeto. Vamos analisar um exemplo prático:

```
{ } loja.json > ...
1   {
2     "loja": {
3       "nome": "Minha Loja de Brinquedos",
4       "localizacao": "Rua Principal, 123, Cidade Feliz",
5       "produtos": [
6         {
7           "id": 1,
8           "nome": "Boneca de Pelúcia",
9           "preco": 29.99,
10          "categoria": "Brinquedos",
11          "estoque": 50
12        },
13        {
14          "id": 2,
15          "nome": "Quebra-Cabeça",
16          "preco": 14.99,
17          "categoria": "Jogos",
18          "estoque": 30
19        },
20        {
21          "id": 3,
22          "nome": "Carrinho de Controle Remoto",
23          "preco": 39.99,
24          "categoria": "Brinquedos",
25          "estoque": 20
26        }
27      ]
28    }
29 }
```



Neste exemplo, temos um objeto que descreve um produto em uma loja de brinquedos. As chaves, como "id", "nome", "preco", "categoria" e "estoque", fornecem informações detalhadas sobre o produto, como seu identificador, nome, preço, categoria e quantidade em estoque.

```
{  
  "id": 1,  
  "nome": "Boneca de Pelúcia",  
  "preco": 29.99,  
  "categoria": "Brinquedos",  
  "estoque": 50  
}
```

Aplicação Prática em APIs

Em um ambiente de desenvolvimento de APIs, o JSON desempenha um papel crucial na troca de informações entre clientes e servidores. Ao fazer uma solicitação a uma API, os clientes recebem respostas em formato JSON, permitindo que os dados sejam facilmente interpretados e utilizados em seus aplicativos. Por exemplo, ao solicitar uma lista de produtos de uma loja de brinquedos por meio de uma API, os dados retornados são frequentemente formatados como objetos JSON contendo detalhes sobre cada produto disponível.

Importância do Conhecimento Profundo do JSON

Para profissionais de desenvolvimento de software, ter um entendimento sólido do JSON é essencial. Dominar a leitura, criação e manipulação de objetos JSON permite que os desenvolvedores construam integrações robustas entre sistemas e serviços externos. Isso resulta em aplicativos mais flexíveis e interoperáveis, capazes de oferecer uma experiência de usuário mais rica e personalizada.



Conclusão: Capacitação para a Inovação

Em resumo, o JSON é uma ferramenta poderosa que impulsiona a interconectividade e a inovação no desenvolvimento de software. Ao compreender profundamente o JSON e sua aplicação em APIs, os profissionais estão equipados para construir soluções mais eficientes e escaláveis, preparando o caminho para o sucesso em um mundo cada vez mais interconectado e orientado a dados.

Awesome API - [Link Awesome API](#)

A "Awesome API" é uma plataforma de dados financeiros que fornece uma ampla gama de informações financeiras e econômicas, incluindo cotações de criptomoedas, ações, índices, moedas estrangeiras, commodities e muito mais. Ela oferece acesso a dados em tempo real e históricos, permitindo que os desenvolvedores construam aplicativos e sistemas que utilizem essas informações para análises, visualizações e outras finalidades.

Na "Awesome API", os dados são transmitidos no formato JSON (JavaScript Object Notation), que é uma escolha comum para a comunicação de dados pela web devido à sua simplicidade, flexibilidade e facilidade de leitura e escrita por humanos e máquinas. Aqui está um exemplo simplificado de como os dados podem ser estruturados em JSON na "Awesome API":

```
"EURBRL": {  
    "code": "EUR",  
    "codein": "BRL",  
    "name": "Euro/Real Brasileiro",  
    "high": "6.8327",  
    "low": "6.8129",  
    "varBid": "-0.0069",  
    "pctChange": "-0.1",  
    "bid": "6.8195",  
    "ask": "6.8221",  
    "timestamp": "1618315093",  
    "create_date": "2021-04-13 08:58:15"  
},
```



As APIs já prontas, como a "Awesome API" que mencionamos, são projetadas especificamente para que seu front-end possa consumir facilmente os dados que elas oferecem. Aqui está como isso funciona:

- **Formato de Dados Padrão:** As APIs geralmente fornecem dados em um formato padrão, como JSON, XML ou outros formatos estruturados. Isso facilita a compreensão e a manipulação dos dados pelo seu front-end.
- **Documentação Clara e Concisa:** As APIs normalmente vêm com uma documentação detalhada que descreve todos os endpoints disponíveis, os parâmetros que podem ser usados em cada solicitação e o formato dos dados retornados. Isso permite que os desenvolvedores do front-end entendam rapidamente como acessar e utilizar os dados fornecidos pela API.
- **Requisições HTTP Simples:** O front-end pode fazer solicitações HTTP diretas para os endpoints da API usando métodos como GET, POST, PUT e DELETE. Essas solicitações são feitas a partir do código JavaScript do front-end, e os dados retornados pela API podem ser facilmente manipulados e exibidos na interface do usuário.
- **Segurança Integrada:** As APIs geralmente incluem mecanismos de segurança, como autenticação e autorização, para proteger os dados e garantir que apenas usuários autorizados possam acessá-los. Isso é importante para garantir a integridade e a segurança dos dados transmitidos entre o back-end e o front-end.

Protocolos HTTP (Hypertext Transfer Protocol):

O HTTP é um protocolo de comunicação utilizado para transferir informações na World Wide Web. Ele define a forma como as mensagens são formatadas e transmitidas entre clientes (como navegadores da web) e servidores web. O HTTP opera no modelo cliente-servidor, onde o cliente faz solicitações e o servidor responde a essas solicitações.

Códigos de Status HTTP:

Os códigos de status HTTP são números de três dígitos que indicam o resultado de uma solicitação HTTP entre o cliente e o servidor. Alguns exemplos comuns de códigos de status incluem:

- **200 OK:** Indica que a solicitação foi bem-sucedida.
- **404 Not Found:** Indica que o recurso solicitado não foi encontrado no servidor.
- **500 Internal Server Error:** Indica que ocorreu um erro interno no servidor ao processar a solicitação.

Documentação dos Códigos de Status do W3Schools:

O W3Schools fornece uma documentação detalhada dos códigos de status HTTP, explicando o que cada código significa e em que contexto ele é normalmente utilizado. Isso é útil para desenvolvedores que estão depurando problemas em suas aplicações web ou que desejam entender melhor as respostas que estão recebendo dos servidores.

Na documentação do W3Schools, você encontrará explicações claras e exemplos de uso para cada código de status, ajudando-o a entender melhor como interpretar e lidar com diferentes situações ao trabalhar com comunicações HTTP em suas aplicações.

Agora vamos ter nosso primeiro contato com um projeto de frontend que utiliza uma API externa pronta, e para isso vamos utilizar o seguinte arquivo HTML:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Pokemon Info</title>
    <link rel="stylesheet" href="style.css" />
  </head>
  <body>
    <div id="container">
      <h1>Pokemon Info</h1>
      <div id="search-container">
        <input
          type="text"
          id="pokemon-input"
          placeholder="Enter Pokemon Name"
        />
        <button id="search-button">Search</button>
      </div>
      <div id="pokemon-info">
        <img id="pokemon-image" src="" alt="Pokemon Image" />
        <h2 id="pokemon-name"></h2>
        <h3 id="pokemon-type"></h3>
        <h3>Abilities:</h3>
        <ul id="pokemon-abilities"></ul>
      </div>
    </div>

    <script src="script.js"></script>
  </body>
</html>
```

Este código HTML cria uma página web simples com a capacidade de pesquisar por Pokémon e exibir informações sobre eles, como nome, tipo e habilidades. As funcionalidades de busca e exibição são implementadas utilizando JavaScript, que é referenciado pelo arquivo "script.js". O estilo da página é definido pelo arquivo "style.css".



Para o arquivo script.js vamos dividir o código em blocos para uma melhor compreensão:

1. Adicionando um Evento para Quando o Documento é Carregado:

```
document.addEventListener("DOMContentLoaded", function () {  
    // Código vai aqui  
});
```

Neste bloco, estamos adicionando um evento que será acionado quando o DOM (Document Object Model) estiver completamente carregado.

2. Atribuindo Elementos do DOM a Variáveis:

```
const pokemonImage = document.getElementById("pokemon-image");  
const pokemonName = document.getElementById("pokemon-name");  
const pokemonAbilities = document.getElementById("pokemon-abilities");  
const pokemonType = document.getElementById("pokemon-type");  
const pokemonInput = document.getElementById("pokemon-input");  
const searchButton = document.getElementById("search-button");
```

Neste bloco, estamos atribuindo elementos específicos do DOM a variáveis para poder manipulá-los mais facilmente posteriormente no código.

3. Adicionando um Evento de Clique ao Botão de Busca:

```
searchButton.addEventListener("click", function () {  
    // Código vai aqui  
});
```

Este bloco adiciona um evento de clique ao botão de busca. Quando o botão é clicado, uma função será executada.

4. Obtendo o Valor do Nome do Pokémon Digitado:

```
const pokemonNameValue = pokemonInput.value.toLowerCase();
```

Neste bloco, estamos obtendo o valor digitado no campo de entrada (input) e convertendo-o para letras minúsculas.

5. Verificando se o Nome do Pokémon é Válido e Chamando a Função de Busca:

```
if (pokemonNameValue.trim() !== "") {  
    searchPokemon(pokemonNameValue);  
}
```

Aqui, estamos verificando se o valor digitado não está vazio (após remover espaços em branco extras). Se não estiver vazio, chamamos a função searchPokemon com o nome do Pokémon como argumento.

6. Função de Busca do Pokémon:

```
function searchPokemon(name) {  
    // Código vai aqui  
}
```

Neste bloco, definimos a função **searchPokemon** que recebe o nome do Pokémon como argumento.

7. Construindo a URL da API e Fazendo uma Requisição:

```
const apiUrl = `https://pokeapi.co/api/v2/pokemon/${name}/`;  
  
fetch(apiUrl)  
    .then((response) => {  
        // Código vai aqui  
    })  
    .catch((error) => {  
        // Código vai aqui  
});
```

Aqui, estamos construindo a URL da API com base no nome do Pokémon fornecido e fazendo uma requisição HTTP usando a função **fetch**. Se a requisição for bem-sucedida, prosseguimos com o próximo bloco. Se houver um erro, o bloco **catch** será executado.

8. Tratamento da Resposta da API:

```
.then((data) => {
    // Código vai aqui
})
```

Neste bloco, estamos tratando a resposta da API após a requisição ser bem-sucedida.

9. Atualizando os Elementos do DOM com os Dados do Pokémon:

```
pokemonImage.src = data.sprites.front_default;
pokemonName.textContent = data.name.toUpperCase();
pokemonType.textContent =
    "Type: " +
    data.types
    .map((type) => type.type.name)
    .join(", ")
    .toUpperCase();
pokemonAbilities.innerHTML = "";
data.abilities.forEach((ability) => {
    const li = document.createElement("li");
    li.textContent = ability.ability.name;
    pokemonAbilities.appendChild(li);
});
```

Aqui, estamos atualizando os elementos do DOM com os dados do Pokémon obtidos da resposta da API.

10. Lidando com Erros:

```
.catch((error) => {
  console.log(error);
  alert("Pokemon not found!");
});
```

Neste bloco, estamos lidando com erros que podem ocorrer durante a requisição da API. Se houver um erro, ele será registrado no console e exibimos um alerta informando que o Pokémon não foi encontrado.

Código completo

```
1  document.addEventListener("DOMContentLoaded", function () {
2    const pokemonImage = document.getElementById("pokemon-image");
3    const pokemonName = document.getElementById("pokemon-name");
4    const pokemonAbilities = document.getElementById("pokemon-abilities");
5    const pokemonType = document.getElementById("pokemon-type");
6    const pokemonInput = document.getElementById("pokemon-input");
7    const searchButton = document.getElementById("search-button");
8
9    searchButton.addEventListener("click", function () {
10      const pokemonNameValue = pokemonInput.value.toLowerCase();
11      if (pokemonNameValue.trim() !== "") {
12        searchPokemon(pokemonNameValue);
13      }
14    });
15
16    function searchPokemon(name) {
17      const apiUrl = `https://pokeapi.co/api/v2/pokemon/${name}/`;
18
19      fetch(apiUrl)
20        .then((response) => {
21          if (!response.ok) {
22            throw new Error("Pokemon not found!");
23          }
24          return response.json();
25        })
26        .then((data) => {
27          pokemonImage.src = data.sprites.front_default;
28          pokemonName.textContent = data.name.toUpperCase();
29          pokemonType.textContent =
30            "Type: " +
31            data.types
32            .map((type) => type.type.name)
33            .join(", ")
34            .toUpperCase();
35          pokemonAbilities.innerHTML = "";
36          data.abilities.forEach((ability) => {
37            const li = document.createElement("li");
38            li.textContent = ability.ability.name;
39            pokemonAbilities.appendChild(li);
40          });
41        })
42        .catch((error) => {
43          console.log(error);
44          alert("Pokemon not found!");
45        });
46      });
47    });
});
```



API TheMealDB

A API do TheMealDB é uma interface de programação de aplicativos que fornece acesso a uma vasta coleção de informações sobre receitas de comida. Aqui está um resumo geral de como ela funciona:

- **Endpoints:** A API possui diferentes endpoints para acessar diferentes tipos de informações, como listagem de categorias de receitas, detalhes de uma receita específica, busca por nome de receita, entre outros.
- **Requisições HTTP:** Para usar a API, você faz requisições HTTP para os endpoints relevantes. Normalmente, você usaria GET para recuperar informações.
- **Formato de Resposta:** As respostas da API geralmente são retornadas em formato JSON, que é um formato de dados comum na web. Isso permite que você facilmente integre os dados em sua aplicação.
- **Autenticação:** Em alguns casos, a API pode exigir autenticação para acessar determinadas informações ou para fins de rastreamento de uso. Isso geralmente é feito através de chaves de API.
- **Manipulação de Dados:** Depois de receber a resposta da API, você pode manipular os dados conforme necessário em sua aplicação. Isso pode incluir exibir receitas, categorias, ingredientes ou outras informações relacionadas à culinária.
- **Limites de Uso:** Como em muitas APIs, pode haver limites de uso, como um número máximo de requisições por período de tempo. Certifique-se de estar ciente desses limites ao integrar a API em sua aplicação.

Vamos utilizar a estrutura abaixo dentro do arquivo index.html:

```
1  <!DOCTYPE html>
2  <html lang="en">
3  |  <head>
4  |  |  <meta charset="UTF-8" />
5  |  |  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6  |  |  <title>App de Receitas</title>
7  |  |  <link rel="stylesheet" href="style.css" />
8  |  </head>
9  |  <body>
10 |  |  <div class="container">
11 |  |  |  <h1>App de Receitas</h1>
12 |  |  |  <div id="meal" class="meal"></div>
13 |  |  |  <button id="get-meal">Obter Receita</button>
14 |  |  </div>
15 |  |  <script src="script.js"></script>
16 |  </body>
17 </html>
```

Agora vamos dividir o nosso código Javascript em blocos, para uma melhor compreensão:

Bloco 1: Adicionar um Event Listener

```
document.getElementById("get-meal").addEventListener("click", function () {  
    // Código aqui  
});
```

Neste bloco, estamos adicionando um event listener ao elemento com o ID "get-meal", que será acionado quando o elemento for clicado.

Bloco 2: Fetching de Dados

```
fetch("https://www.themealdb.com/api/json/v1/1/random.php")  
    .then((response) => response.json())  
    .then((data) => {  
        // Código aqui  
    })  
    .catch((error) => {  
        console.log("Erro ao obter receita:", error);  
    });
```

Este bloco realiza uma solicitação de busca (fetch) para uma API de receitas aleatórias. Se a solicitação for bem-sucedida, os dados são convertidos para o formato JSON e processados.



Bloco 3: Manipulação de Dados

```
const meal = data.meals[0];
const mealDiv = document.getElementById("meal");
mealDiv.innerHTML =
  <h2>${meal.strMeal}</h2>
  
  <h3>Ingredientes</h3>
  <ul>
    ${getIngredients(meal).join("")}
  </ul>
  <h3>Instruções</h3>
  <p>${meal.strInstructions}</p>
  `;
```

Neste bloco, os dados da refeição são extraídos do objeto **data** e inseridos na página HTML. Isso inclui o nome da refeição, uma imagem, a lista de ingredientes e as instruções.

Bloco 4: Função getIngredients

```
function getIngredients(meal) {
    const ingredients = [];

    for (let i = 1; i <= 20; i++) {
        if (meal[`strIngredient${i}`]) {
            ingredients.push(`

                <li>
                    ${meal[`strIngredient${i}`]} - ${meal[`strMeasure${i}`]}
                </li>
            `);
        } else {
            break;
        }
    }

    return ingredients;
}
```

Esta função é responsável por extrair os ingredientes da refeição e suas respectivas quantidades do objeto **meal** e retorná-los em uma lista HTML.



Código Completo:

```
1  document.getElementById("get-meal").addEventListener("click", function () {
2    fetch("https://www.themealdb.com/api/json/v1/1/random.php")
3      .then((response) => response.json())
4      .then((data) => {
5        const meal = data.meals[0];
6        const mealDiv = document.getElementById("meal");
7        mealDiv.innerHTML =
8          `

## ${meal.strMeal}


9          
10         <h3>Ingredientes</h3>
11         <ul>
12           ${getIngredients(meal).join("")}
13         </ul>
14         <h3>Instruções</h3>
15         <p>${meal.strInstructions}</p>
16       `;
17     })
18     .catch((error) => {
19       console.log("Erro ao obter receita:", error);
20     });
21   });
22 }
```

```
23   function getIngredients(meal) {
24     const ingredients = [];
25
26     for (let i = 1; i <= 20; i++) {
27       if (meal[`strIngredient${i}`]) {
28         ingredients.push(`<li> ${meal[`strIngredient${i}`]} - ${meal[`strMeasure${i}`]}</li>`);
29       } else {
30         break;
31       }
32     }
33
34     return ingredients;
35   }
36 }
```

Módulo 12

DEPURAÇÃO E DESENVOLVIMENTO EFICIENTE

DEPURAÇÃO E DESENVOLVIMENTO EFICIENTE

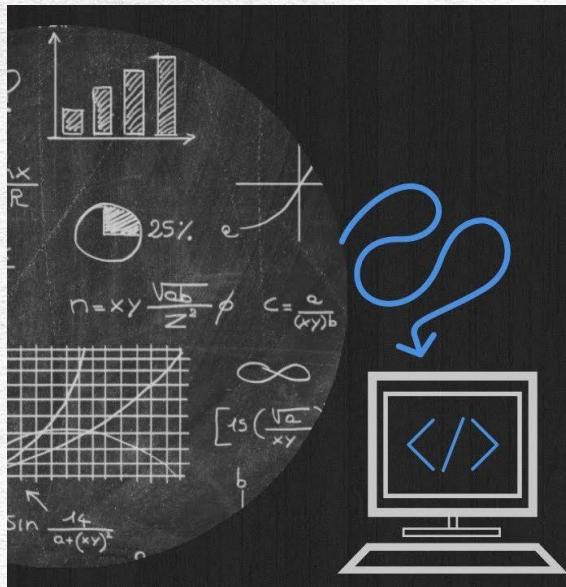
DEPURAÇÃO E DESENVOLVIMENTO EFICIENTE



Começaremos a partir deste módulo, a escrever códigos com palavras em inglês. Não se preocupe caso não tenha conhecimento da língua inglesa, apenas iremos adotar como boa prática e padronização dos nossos códigos.

Será muito comum na sua vida de programador se deparar com código em inglês, então iniciaremos essa prática a partir de agora.

Explicaremos tudo no decorrer das próximas aulas, para você continuar a construir um conhecimento sólido e utilizável no mundo da programação hoje.



Nesse módulo continuaremos com alguns conceitos da matemática, por que tanto a Matemática quanto a Programação contam com conteúdo específicos que contribuem para o aperfeiçoamento do raciocínio lógico nos estudantes.

Você precisa ser muito bom em matemática para ser um programador ? A resposta é NÃO.

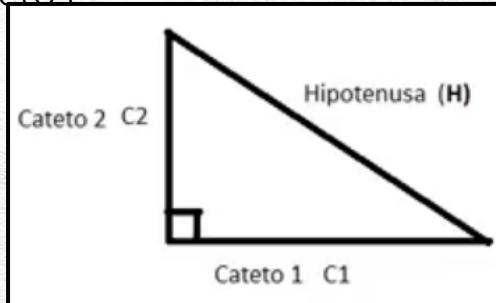
Mas ter o conhecimento da matemática básica é essencial para alcançar um bom aprendizado e para desenvolver um pensamento mais rápido e ágil.



Módulo 12 – Os vários tipos de erros (2 / 8)

Iremos explorar o teorema de Pitágoras para criarmos um código que calcule ele. Vamos então relembrar como que na matemática esse teorema é feito.

O teorema de Pitágoras é uma relação entre as medidas dos lados de um triângulo retângulo, conhecidos como hipotenusa e catetos:



Fórmula matemática que representa o teorema de Pitágoras.

$$H = \sqrt(C1^2 + C2^2)$$

Será esta fórmula que iremos aplicar ao nosso código. Além de utilizarmos a função que calcula a raiz quadrada criada nas aulas passadas, porém a sua nomenclatura será em inglês, mas se reparar é o mesmo código já criado.

```
js calculatePythagoras.js > ...
1  function calculateSquareRoot(radicand) {
2    return radicand ** (1 / 2);
3 }
```

IGUAL
↔

```
3  function calcularRaizQuadrada(base) {
4    return base ** (1 / 2);
5 }
```

Lembrando uma função é uma caixa que guarda tarefas específicas que iremos executar em algum momento no nosso código, no caso acima, calcular a raiz quadrada de um número.

Vamos entender como a nossa função irá aplicar a equação matemática e calculará o teorema de Pitágoras através do código em JavaScript.

$$H = \sqrt{C1^2 + C2^2}$$



```
js calculatePythagoras.js > ⚡ calculatePythagoras
1  function calculateSquareRoot(radicand) {
2    return radicand ** (1 / 2);
3  }
4
5  function calculatePythagoras(side1, side2) [
6    const sum = side1 ** 2 + side2 ** 2;
7    return calculateSquareRoot(sum);
8  ]
```

Declaramos a nossa **função** e chamaremos de **calculatePythagoras** (calcular Pitágoras).

Para calcularmos o teorema de Pitágoras, nossa função precisa conhecer os valores dos catetos para ser executada.

Esses valores serão os **parâmetros** da nossa função **side1** e **side2** (side = lado).

Declaramos a **variável sum** (soma) para guardar a adição dos catetos ao quadrado.

E em seguida, retornaremos o valor da nossa função que calcula a raiz quadrada, **calculateSquareRoot**, chamando como argumento a nossa **variável sum**, já que ela armazenou o **valor da equação (side1 ** 2 + side2 ** 2)**

Para continuarmos o nosso código, precisamos entender que a boa prática da escrita em inglês, refere-se ao código em si, ou seja, os códigos em JavaScript em inglês são importantes para a leitura e compreensão dos **Programadores**, seja você, um colega de equipe ou outro programador qualquer que tenha acesso ao programa.

Já as mensagens que vamos exibir na tela, podemos escrever em português, pois essa mensagem é importante para quem for usar o seu programa, o **Usuário**.

Simplificando, o Programador analisa e escreve o código, já o Usuário usa o produto daquele código.

```
JS calculatePythagoras.js > ...
1  function calculateSquareRoot(radical) {
2    return radical ** (1 / 2);
3  }
4
5  function calculatePythagoras(side1, side2) {
6    const sum = side1 ** 2 + side2 ** 2;
7    return calculateSquareRoot(sum);
8  }
9
10 console.log(`O tamanho da hipotenusa de um triângulo retângulo de lados 3 e 4 é ${calculatePythagoras(3, 4)}`);
```

Interação do Programador

Interação do Usuário

Adicionamos a mensagem a ser impressa na tela, fazendo o chamado da **função calculatePythagoras** e passando os **valores 3 e 4** como **Argumento**.

Assim, quando executarmos nosso programa, ele irá caminhar por todas as funções chamadas e retornará o seguinte resultado:

```
PS C:\Users\danie\JavaScript Impressionador\modulo_4\aula4_1> node ./calculatePythagoras.js
A hipotenusa de um triangulo de lados 3 e 4 é 5
```

O nosso programa funcionou e retornou o resultado esperado. Mas vamos imaginar que o nosso **programa NÃO funcione**. Em quais cenários isso poderia acontecer?

Se a chamada da nossa função fosse feita com o nome errado, isso geraria um **erro** do tipo **ReferenceError** (erro de Referência):

```
console.log(`A hipotenusa de um triângulo de lados 3 e 4 é: ${calculate3Pythagoras(3, 4)}`);
```

```
PS C:\Users\danie\JavaScript Impressionador\modulo_4\aula4_1> node ./calculatePythagoras.js
C:\Users\danie\JavaScript Impressionador\modulo_4\aula4_1\calculatePythagoras.js:11
  `A hipotenusa de um triangulo de lados 3 e 4 é: ${calculat3ePythagoras(3, 4)}`
  ^
ReferenceError: calculat3ePythagoras is not defined
    at Object.<anonymous> (C:\Users\danie\JavaScript Impressionador\modulo_4\aula4_1\calculatePythagoras.js:11:3)
    at Module._compile (node:internal/modules/cjs/loader:1218:14)
    at Module._extensions..js (node:internal/modules/cjs/loader:1272:10)
    at Module.load (node:internal/modules/cjs/loader:1081:32)
    at Module._load (node:internal/modules/cjs/loader:922:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:81:12)
    at node:internal/main/run_main_module:23:47

Node.js v18.13.0
```



Neste momento, falaremos sobre os possíveis **erros** que um programa pode gerar. De modo geral, quando você faz algo errado no código, você encontrará possíveis erros.

Existem os **erros de sintaxe**, que são erros de ortografia, digitação, erros de Javascript, como o **ReferenceError**. Ele é um erro de nomenclatura, já que chamamos nossa função com o nome errado. Esse erro nos indica que ele **não encontrou a Referência** da função, ou seja, não encontrou o nome dela.

Temos um outro exemplo muito comum, que é o **TypeError**, um erro que ocorre quando uma operação não pode ser executada, normalmente, quando **um valor não é o tipo** esperado ou uma **atribuição de valor não pode ser feita**, como o exemplo a seguir:

```
function calculatePythagoras(side1, side2) {  
    const sum = side1 ** 2 + side2 ** 2;  
    sum = calculateSquareRoot(sum);  
    return sum;  
}
```

```
C:\Users\danie\JavaScript Impressionador\modulo_4\aula4_1\calculatePythagoras.js:7  
    sum = calculateSquareRoot(sum);  
           ^  
  
TypeError: Assignment to constant variable.  
    at calculatePythagoras (C:\Users\danie\JavaScript Impressionador\modulo_4\aula4_1\calculatePythagoras.js:7:7)  
    at Object.<anonymous> (C:\Users\danie\JavaScript Impressionador\modulo_4\aula4_1\calculatePythagoras.js:12:53)  
    at Module._compile (node:internal/modules/cjs/loader:1218:14)  
    at Module._extensions..js (node:internal/modules/cjs/loader:1272:10)  
    at Module.load (node:internal/modules/cjs/loader:1081:32)  
    at Module._load (node:internal/modules/cjs/loader:922:12)  
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:81:12)  
    at node:internal/main/run_main_module:23:47
```

Esse erro é gerado porque tentamos atribuir um novo valor a uma variável do **tipo constante**, se fosse do **tipo let** não teríamos esse erro, pois ela pode ter seu valor alterado, enquanto uma **variável tipo const** não.

Outro tipo de erro que podemos encontrar nos nossos códigos, são os **erros lógicos**. Que são erros onde a sintaxe do programa está correta, então ao executarmos ele não gerará nenhuma mensagem de erro, mas o código não está executando com o resultado correto.

Eles são erros mais difíceis de serem corrigidos, pois como não há mensagem de erro resultando para nos direcionar, demoramos mais para identificá-los.

Vamos exemplificar, imagine que durante a criação do nosso programa que calcula o Pitágoras, esquecêssemos de aplicar a raiz quadrada:

```
JS calculatePythagoras.js > ...
1  function calculateSquareRoot(radicand) {
2    return radicand ** (1 / 2);
3  }
4
5  function calculatePythagoras(side1, side2) {
6    const sum = side1 ** 2 + side2 ** 2;
7    return sum;
8  }
9
10 console.log(
11   `A hipotenusa de um triangulo de lados 3 e 4 é: ${calculatePythagoras(3, 4)}`
12 );
```

```
PS C:\Users\danie\JavaScript Impressionador\modulo_4\aula4_1> node ./calculatePythagoras.js
A hipotenusa de um triangulo de lados 3 e 4 é: 25
```

Analisando o código ao lado, vemos que o programa executa sem que mensagens de erro aparecem, então é **um programa que está funcionando**.

Mas sua **lógica está incorreta**, retornando o **resultado errado** de **25**, enquanto que esperamos o **valor 5**.

Isso ocorre porque retiramos o uso da função que calcula a raiz quadrada, e apenas retornamos a equação de soma que está na variável sum.



Nessa aula apresentamos de uma maneira simplificada sobre alguns **tipos de erros** que nos deparamos na programação.

Os erros existem em todas as linguagens de programação, e no JavaScript não seria diferente, eles possuem vários nomes: **bugs, erros, exceções**, entre outros.

Os **erros referem-se a situações que não permite que um programa funcione** normalmente, e empurram o programa para lançar erros ao usuário. O programa irá coletar o máximo de informações possíveis sobre o erro e então identificará que não pode avançar.

A lei de Murphy diz que tudo que pode dar errado eventualmente dará errado. E isso se aplica demais ao mundo da programação.

Erros no JavaScript são problemas muito comuns, e começar a **entender, analisar, corrigir e tratar** esses erros, são tarefas muito importantes e comuns para o programador.

Existem diversos tipos de erros e você como programador deve se preocupar em saber como identificar e lidar com eles de forma eficiente, este módulo te ajudará nesse processo.

Na aula passada, começamos a falar um pouco sobre erros e aprofundaremos sobre esse assunto entendendo que é muito comum nos depararmos com erros ao programar, afinal, só não erra código, quem não escreve código.

E por ser algo tão comum de acontecer, precisamos aprender de forma inteligente a resolve-los.

Utilizamos o termo bug quando algum erro ocorre no nosso programa, é um jargão da informática, que são as falhas inesperadas que ocorrem quando tentamos executar um código, que geram comportamentos incorretos ou inesperados.



Vamos gerar alguns erros no programa que desenvolvemos anteriormente, para iniciarmos uma primeira abordagem de como lidar com eles.

Começando com o seguinte cenário:

Na função **calculateSquareRoot** ao invés de elevarmos nossa **variável radicand** a **potência (**) de $\frac{1}{2}$** , multiplicarmos (*) o **radicand** por $\frac{1}{2}$.

A função **calculatePythagoras**, também fosse feito essa troca na equação que a **variável sum** está armazenando e o **retorno** da função fosse um **texto "sum"**, ao invés da variável.

Código CORRETO

```
function calculateSquareRoot(radicand) {  
    return radicand **(1 / 2);  
}  
  
function calculatePythagoras(side1, side2) {  
    const sum = side1 **2 + side2 ** 2;  
    return calculateSquareRoot(sum);  
}  
  
console.log(  
    `A hipotenusa de um triangulo de lados 3 e 4 é: ${calculatePythagoras(3, 4)}`  
);
```

Código com ERROS

```
function calculateSquareRoot(radicand) {  
    return radicand *(1 / 2);  
}  
  
function calculatePythagoras(side1, side2) {  
    const sum = side1 *2 + side2 ** 2;  
    return calculateSquareRoot('sum');  
}  
  
console.log(  
    `A hipotenusa de um triangulo de lados 3 e 4 é: ${calculatePythagoras(3, 4)}`  
);
```

PS C:\Users\danie\JavaScript Impressionador\modulo_4\aula4_1> node ./calculatePythagoras.js
A hipotenusa de um triangulo de lados 3 e 4 é: 5

PS C:\Users\danie\JavaScript Impressionador\modulo_4\aula4_1> node ./calculatePythagoras.js
A hipotenusa de um triangulo de lados 3 e 4 é: NaN

Rpare que o código com erros, não retorna nenhuma mensagem de erro, como vimos na aula passada. Mas um ponto de atenção é o seu retorno: **Nan**.

Propriedade global **NaN**

NaN é um tipo especial de retorno, que significa **Not-A-Number**, ou seja, não é um número.

Ele é retornado quando tentamos transformar uma string em um número ou quando uma operação de matemática falha.

É um símbolo que você tentou tratar como número algo que não é de fato um número.

Analizando o nosso código, identificamos que realmente tentamos usar um **texto 'sum'** como **argumento** para a **função calculateSquareRoot** que espera um **número** para realizar uma operação de matemática.

```
function calculateSquareRoot(radicand) {
  return radicand * (1 / 2);
}

function calculatePythagoras(side1, side2) {
  const sum = side1 * 2 + side2 ** 2;
  return calculateSquareRoot('sum');
}

console.log(
  'A hipotenusa de um triangulo de lados 3 e 4 é: ${calculatePythagoras(3, 4)}'
);
```

Mas vamos imaginar que estivesse difícil de identificar esse erro no nosso código. Quais ferramentas podemos utilizar para nos auxiliar a encontrar esse erros?



Uma das ferramentas mais simples que podemos utilizar para verificar se as nossas funções, variáveis, programas estão retornando o resultado esperado, é o nosso comando `console.log()`.
No exemplo a seguir, iremos utilizá-lo para verificar passo a passo das nossas funções:

```
function calculatePythagoras(side1, side2) {
    console.log(
        'Os tamanhos dos catetos do meu triângulo retângulo são: ',
        side1,
        side2
    );
    const sum = side1 * 2 + side2 ** 2;
    return calculateSquareRoot('sum');
}

console.log(
    'A hipotenusa de um triangulo de lados 3 e 4 é: ${calculatePythagoras(3, 4)}'
);
```

```
PS C:\Users\danie\JavaScript Impressionador\modulo_4\aula4_1> node ./calculatePythagoras.js
Os tamanhos dos catetos do meu triângulo retângulo são: 3 4
A hipotenusa de um triangulo de lados 3 e 4 é: NaN
```

Adicionamos a nossa **função calculatePythagoras**, um **console.log** para imprimir os valores das variáveis que a função está recebendo como parâmetro.

Percebe que nesse código, estamos utilizando um outro formando de escrita de texto que não é a CONCATENAÇÃO e nem a String Template.
Eles existem

Já que o **console.log** é uma função, ele sabe se comportar com números diferentes de parâmetros, sejam textos, variáveis, funções.

Nessa situação, ele recebe um parâmetro do tipo texto e mais dois parâmetros, sendo as variáveis `side1` e `side2`.

Veja que ele **imprimiu todos os console.log** do código. E verificamos que a nossa função está recebendo os **valores numéricos 3 e 4**, ou seja, não é nesse local que o nosso **retorno NaN** está sendo gerado.

Vamos acrescentar mais uma etapa em nosso código para continuarmos verificando se os retornos são o que esperamos.

Desta vez, colocamos um `console.log` para retornar o valor da nossa variável `sum`, que está armazenando o valor de uma equação. Como sabemos que a nossa variável `side1` possui valor de 3 e `side2` valor de 4, esperamos que o retorno de `sum`, na lógica correta ($3^2 + 4^2$), seja 25.

```
function calculatePythagoras(side1, side2) {
  console.log(
    'Os tamanhos dos catetos do meu triângulo retângulo são: ',
    side1,
    side2
  );
  const sum = side1 * 2 + side2 ** 2;
  console.log('O valor da variável sum é: ', sum);
  return calculateSquareRoot('sum');
}
```

```
PS C:\Users\danie\JavaScript Impressionador\modulo_4\aula4_1> node ./calculatePythagoras.js
Os tamanhos dos catetos do meu triângulo retângulo são:  3 4
A hipotenusa de um triangulo de lados 3 e 4 é: NaN
PS C:\Users\danie\JavaScript Impressionador\modulo_4\aula4_1> node ./calculatePythagoras.js
Os tamanhos dos catetos do meu triângulo retângulo são:  3 4
[0 valor da variável sum é:  22
A hipotenusa de um triangulo de lados 3 e 4 é: NaN]
```

Aqui já identificamos um erro e o `console.log` nos dá indicação de onde pode estar, pois o **retorno não é o esperado**. Nossa função está retornando 22, ou seja, observando a estrutura da nossa função, conseguimos identificar um **erro de sintaxe** na nossa equação.

Dessa forma, conseguimos utilizar o console.log como estratégia para definir blocos de execução, de um console.log para outro. E assim encontrar com mais facilidade os possíveis erros.

Durante o nosso primeiro console.log, verificamos que o retorno era o esperado.

No segundo console.log, o retorno estava incorreto.

Com essas duas informações, conseguimos traçar em qual local o erro está, que seria entre eles.

No caso ao lado, o erro de sintaxe está na linha da nossa variável sum.

Nesse exemplo, utilizamos o **console.log** como **ferramenta para mapear nossa função calculatePythagoras, analisar seus retornos, identificar os possíveis erros e corrigi-los.**

```
function calculatePythagoras(side1, side2) {
    console.log(
        'Os tamanhos dos catetos do meu triângulo retângulo são: ', 
        side1,
        side2
    );
    const sum = side1 * 2 + side2 ** 2; X - local do erro
    console.log('O valor da variável sum é: ', sum);
    return calculateSquareRoot('sum');
}
```

Após corrigir o primeiro erro localizado, executamos novamente nosso programa. E verificamos que a correção, **não alterou ainda o nosso retorno final NaN**.

Então, repetiremos esse processo, incluindo os `console.log` em outras etapas do nosso código até conseguirmos encontrar o local do erro.

```
function calculateSquareRoot(radicand) {
    console.log('O valor do meu radicando é: ', radicand); ←
    return radicand * (1 / 2);
}

function calculatePythagoras(side1, side2) {
    console.log(
        'Os tamanhos dos catetos do meu triângulo retângulo são: ',
        side1,
        side2
    );
    const sum = side1 ** 2 + side2 ** 2;
    console.log('O valor da variável sum é: ', sum);
    return calculateSquareRoot('sum');
}

console.log(
    'A hipotenusa de um triângulo de lados 3 e 4 é: ${calculatePythagoras(3, 4)}'
);
```

```
const sum = side1 ** 2 + side2 ** 2;
```

```
PS C:\Users\danie\JavaScript Impressionador\modulo_4\aula4_1> node ./calculatePythagoras.js
Os tamanhos dos catetos do meu triângulo retângulo são: 3 4
O valor da variável sum é: 25
A hipotenusa de um triangulo de lados 3 e 4 é: NaN
```

programação. Eles existem em todos eles, a gente tem vários nomes para eles: bugs, erros, exceções, etc.

A próxima etapa do nosso código é a nossa função **calculatePythagoras**, executar a função **calculateSquareRoot**. Então iremos adicionar um `console.log` para verificar essa função.

Lembrando que quando a nossa função **calculateSquareRoot** é executada, o JavaScript passa o valor do parâmetro, resolve a função e retorna o **seu resultado como argumento dentro do retorno da função calculatePythagoras**.

Opa! O retorno do novo console.log é "O valor do meu radicando é: sum"!

Repare aqui o valor de retorno esperado deveria ser um número, especificamente o valor 25 da nossa equação armazenada na nossa variável sum, e não a palavra "sum".

Pronto! Analisando esse retorno, conseguimos encontrar que o erro do nosso código está no argumento da função calculateSquareRoot, que está sendo executada dentro da nossa função calculatePythagoras.

```
PS C:\Users\danie\JavaScript Impressionador\modulo_4\aula4_1> node ./calculatePythagoras.js
Os tamanhos dos catetos do meu triângulo retângulo são: 3 4
O valor da variável sum é: 25
O valor do meu radicando é: sum
A hipotenusa de um triangulo de lados 3 e 4 é: NaN
```

Erros são provavelmente o principal constrangimento de qualquer linguagem de programação. Eles existem em todas elas, e muitas vezes variam de acordo com o idioma.

```
function calculateSquareRoot(radicand) {
    console.log('O valor do meu radicando é: ', radicand);
    return radicand * (1 / 2);
}

function calculatePythagoras(side1, side2) {
    console.log(
        'Os tamanhos dos catetos do meu triângulo retângulo são: ',
        side1,
        side2
    );
    const sum = side1 ** 2 + side2 ** 2;
    console.log('O valor da variável sum é: ', sum);
    return calculateSquareRoot('sum');
}

console.log(
    'A hipotenusa de um triangulo de lados 3 e 4 é: ${calculatePythagoras(3, 4)}'
);
```

Corrigindo novamente nosso, deparamos ainda com mais um erro de sintaxe. O nosso retorno final, deixou de ser um tipo NaN, e passou a ser um valor numérico 12.5 , mas o esperado é o valor 5.

Repetiremos mais uma vez o processo até que o nosso programa funcione com os retornos esperados.

```
function calculateSquareRoot(radicand) {  
    console.log('O valor do meu radicando é: ', radicand);  
    return radicand * (1 / 2);  
}
```

Vamos corrigir o último erro encontrado na função calculateSquareRoot.

E executar novamente todo o nosso programa!

```
    | return calculateSquareRoot(sum);
```

```
PS C:\Users\danie\JavaScript Impressionador\modulo_4\aula4_1> node ./calculatePythagoras.js  
Os tamanhos dos catetos do meu triângulo retângulo são: 3 4 OK  
O valor da variável sum é: 25 OK  
O valor do meu radicando é: 25 OK  
A hipotenusa de um triangulo de lados 3 e 4 é: 12.5
```

Erros são provavelmente o principal constrangimento de qualquer linguagem de programação. Eles existem em todas elas, e muitos devolvendo nomes para eles: bugs, erros, exceções, etc.

```
function calculateSquareRoot(radicand) {  
    console.log('O valor do meu radicando é: ', radicand);  
    return radicand ** (1 / 2);  
}  
  
function calculatePythagoras(side1, side2) {  
    console.log(  
        'Os tamanhos dos catetos do meu triângulo retângulo são: ',  
        side1,  
        side2  
    );  
    const sum = side1 ** 2 + side2 ** 2;  
    console.log('O valor da variável sum é: ', sum);  
    return calculateSquareRoot(sum);  
}  
  
console.log(  
    `A hipotenusa de um triangulo de lados 3 e 4 é: ${calculatePythagoras(3, 4)}`  
);
```

Agora sim! Nosso programa além de funcionar perfeitamente, está retornando os valores corretos.

```
PS C:\Users\danie\JavaScript Impressionador\modulo_4\aula4_1> node ./calculatePythagoras.js
Os tamanhos dos catetos do meu triângulo retângulo são: 3 4
O valor da variável sum é: 25
O valor do meu radicando é: 25
A hipotenusa de um triangulo de lados 3 e 4 é: 5
```

Erros são provavelmente o principal construto de qualquer linguagem de programação. Eles existem em todas elas, a gente tem vários nomes para eles: bugs, erros, exceções, etc.

Com o conteúdo de hoje, nos aproximamos um pouco mais sobre como encontrar, analisar e corrigir erros de uma maneira simples.

Mas devemos lembrar que o uso do `console.log` como ferramenta, servirá para você programador compreender e mapear seu código de possíveis erros. Com o programa rodando perfeitamente, temos que ter a boa prática de não deixarmos os nosso `console.log` para serem executados pelos usuários.

O JavaScript é uma linguagem que originalmente foi criado para funcionar em conjunto com as tecnologias que permitiam fazer páginas de internet, uma tecnologia que foi criada para criar uma internet inteligente.

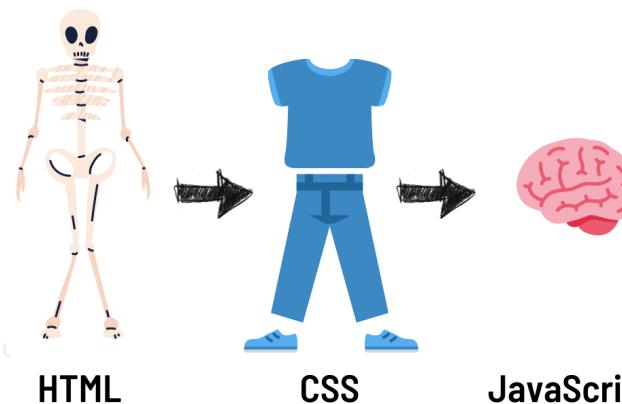
Saber lidar com problemas, identificar, rastrear e resolver lidando com sistemas que rodam na internet também faz parte do dia-a-dia do desenvolvedor de Javascript.

Nesse momento vamos iniciar os primeiros passos em algumas **tecnologia Web**, que atuam junto com o Javascript. Então vamos realizar algumas modificações no programa que desenvolvemos sobre o Teorema de Pitágoras.

Não se preocupe se não entender todo o conteúdo dessas tecnologias, pois iremos continuar abordando um pouco mais sobre elas no decorrer do curso.

Em um **projeto Web**, existem **3 tecnologias** que são principais para seu desenvolvimento, o **HTML**, o **CSS** e o nosso **JavaScript**.

O **HTML** terá como objetivo definir a estrutura da página da internet, enquanto o **CSS** é responsável pela aparência das nossas páginas, define o estilo, posicionamento, cores, entre outras. E o **JavaScript** adiciona a inteligência do nosso projeto.



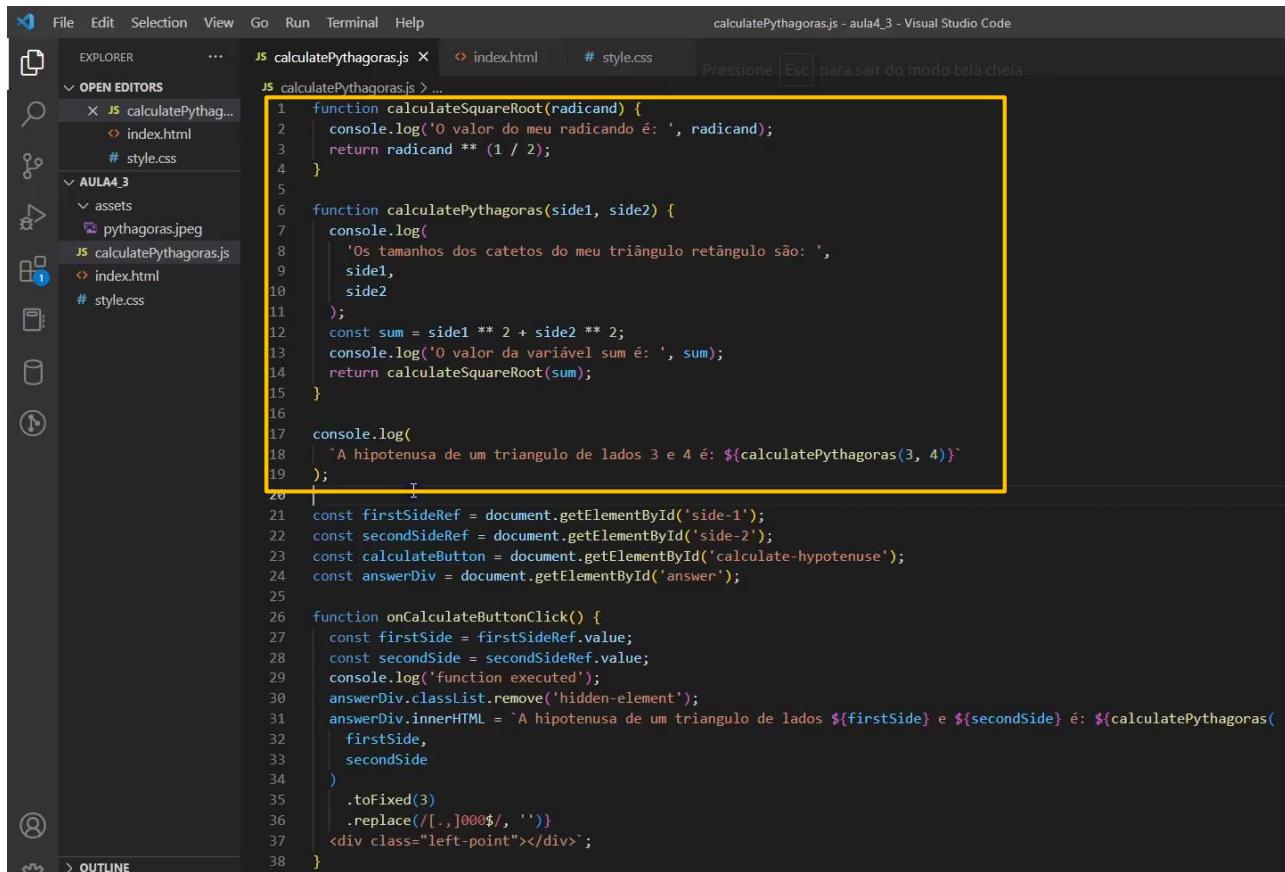
Erros são provavelmente o principal construto de qualqu

elas, a gente tem vários nomes para eles: bugs, erros, exceções, etc



DESENVOLVIMENTO WEB - FULL STACK IMPRESSIONADOR I HASHTAG PROGRAMAÇÃO

Módulo 12 – Dentro do Navegador (2 /19)



```

File Edit Selection View Go Run Terminal Help
calculatePythagoras.js - aula4_3 - Visual Studio Code
Pressione Esc para sair do modo tela cheia

JS calculatePythagoras.js > ...
OPEN EDITORS
AULAS
Assets
pythagoras.jpeg
JS calculatePythagoras.js
index.html
# style.css

1 function calculateSquareRoot(radicand) {
2   console.log('O valor do meu radicando é: ', radicand);
3   return radicand ** (1 / 2);
4 }

5

6 function calculatePythagoras(side1, side2) {
7   console.log(
8     'Os tamanhos dos catetos do meu triângulo retângulo são: ',
9     side1,
10    side2
11  );
12  const sum = side1 ** 2 + side2 ** 2;
13  console.log('O valor da variável sum é: ', sum);
14  return calculateSquareRoot(sum);
15 }

16

17 console.log(
18   'A hipotenusa de um triangulo de lados 3 e 4 é: ${calculatePythagoras(3, 4)}'
19 );

20

21 const firstSideRef = document.getElementById('side-1');
22 const secondSideRef = document.getElementById('side-2');
23 const calculateButton = document.getElementById('calculate-hypotenuse');
24 const answerDiv = document.getElementById('answer');

25

26 function onCalculateButtonClick() {
27   const firstSide = firstSideRef.value;
28   const secondSide = secondSideRef.value;
29   console.log('function executed');
30   answerDiv.classList.remove('hidden-element');
31   answerDiv.innerHTML = `A hipotenusa de um triangulo de lados ${firstSide} e ${secondSide} é: ${calculatePythagoras(
32     firstSide,
33     secondSide
34   ) .toFixed(3)
35   .replace(/[,]000$/, '')}`;
36   <div class="left-point"></div>;
37 }
38 }

```

Os arquivos de HTML e CSS, estão disponíveis para download na plataforma do curso, então você não precisa se preocupar em criá-los.

Quando criamos um arquivo JavaScript adicionamos após o nome a extensão **.js**, o arquivo HTML terá a extensão **.html** e o arquivo CSS a extensão **.css**, ou seja, ao criar um arquivo nomeando e adicionando a extensão você automaticamente cria um arquivo específico dessas tecnologias.

Com essas informações, partiremos para o nosso programa. Repare que estrutura inicial do nosso programa está idêntica ao que viemos construindo até agora.

E as informações novas estão a partir da linha 21, e vamos entender-las a partir de agora.

```
21 const firstSideRef = document.getElementById('side-1');
22 const secondSideRef = document.getElementById('side-2');
23 const calculateButton = document.getElementById('calculate-hypotenuse');
24 const answerDiv = document.getElementById('answer');
```

Precisamos entender que uma página da internet é composta por botões, campos para escrever textos, que são elementos que o usuário pode interagir. A estrutura de código das linhas acima possuem a lógica de armazenar dentro de variáveis as referências desses elementos para adicionar ações a eles.

Logo abaixo, definimos uma ação para um botão através de um botão de click, está ação está armazenada dentro de uma função.

```
26 function onCalculateButtonClick() {
27   const firstSide = firstSideRef.value;
28   const secondSide = secondSideRef.value;
29   console.log('function executed');
30   answerDiv.classList.remove('hidden-element');
31   answerDiv.innerHTML = `A hipotenusa de um triangulo de lados ${firstSide} e ${secondSide} é: ${calculatePythagoras(
32     firstSide,
33     secondSide
34   )}
35     .toFixed(3)
36     .replace(/[,]000$/, '')
37   <div class="left-point"></div>`;
38 }
```

Por fim, adicionamos essa ação ao botão de click, ou seja, executamos essa função no momento em que houver um click de fato.

```
40 calculateButton.addEventListener('click', onCalculateButtonClick);
```

Assim como no nosso terminal via Node.js o JavaScript executa os programa retornando as ações que pedimos para executar.

Quando unimos HTML, CSS e o JavaScript, iremos utilizar o **NAVEGADOR** para executar nossos programas.

O comportamento do navegador será abrir uma página da internet e carregar o arquivo HTML , ou seja, aplicar a estrutura que criamos para nossa página, aplicar a aparência para essas estruturas criadas no nosso arquivo CSS.

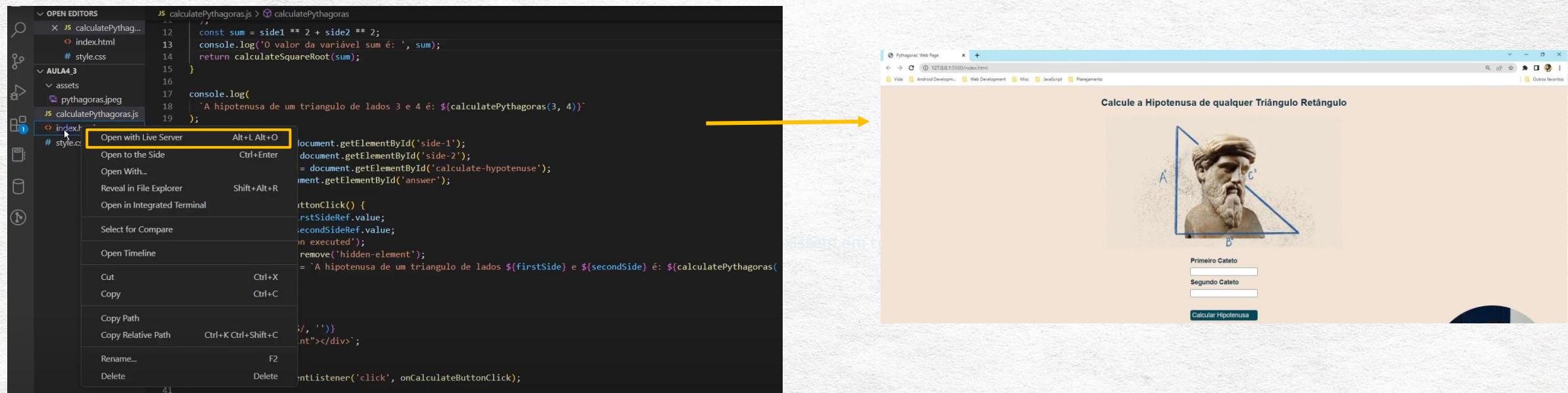
Dentro do nosso arquivo HTML , teremos a **referência do nosso arquivo JavaScript**, isso quer dizer, que o navegador sabe que precisa ler esse arquivo (de cima para baixo) e ele executa o que está no nosso código.

E sempre irá associar essas referência a eventos que estão ocorrendo no nosso programa, como ao clicar em algo uma ação irá ser feita.

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8" />
5      <meta http-equiv="X-UA-Compatible" content="IE=edge" />
6      <meta name="viewport" content="width=device-width, initial-scale=1.0" />
7      <title>Pythagoras' Web Page</title>
8      <link rel="stylesheet" href="style.css" /> CS
9  </head>
10 <body> S
11     <h2>Calcule a Hipotenusa de qualquer Triângulo Retângulo</h2>
12     <div class="answer-container">
13         
18         <div id="answer" class="hidden-element"></div>
19     </div>
20     <div class="inputs-container">
21         <label for="side-1">Primeiro Cateto</label>
22         <input id="side-1" type="text" />
23         <label for="side-2">Segundo Cateto</label>
24         <input id="side-2" type="text" />
25         <button id="calculate-hypotenuse">Calcular Hipotenusa</button>
26     </div>
27     <script src="/calculatePythagoras.js"></script> JAVASCRIPT
28 </body>
29 </html>
```

Vamos agora **abrir nosso programa** dentro do **navegador**, para isso, iremos utilizar uma extensão do VS Code , o Live Server (caso não lembre como baixar extensões no VS Code, você poderá rever seu conteúdo na página 40).

Iremos clicar com o botão esquerdo do mouse no arquivo do **index.html**, e clicar em **Open with Live Server** ou o atalho Alt +L Alt + O. Com isso abriremos o nosso o programa dentro do navegador.



Como aprendemos a identificar alguns erros dentro do nosso terminal, no nosso navegador não será diferente. Estrategicamente o nosso código possui alguns erros para aprender a identificar. Mas primeiramente, iremos entender como a nossa página de internet está funcionando.

Calcule a Hipotenusa de qualquer Triângulo Retângulo

A hipotenusa de um triângulo de lados 3 e 4 é: 5

Primeiro Cateto
3

Segundo Cateto
4

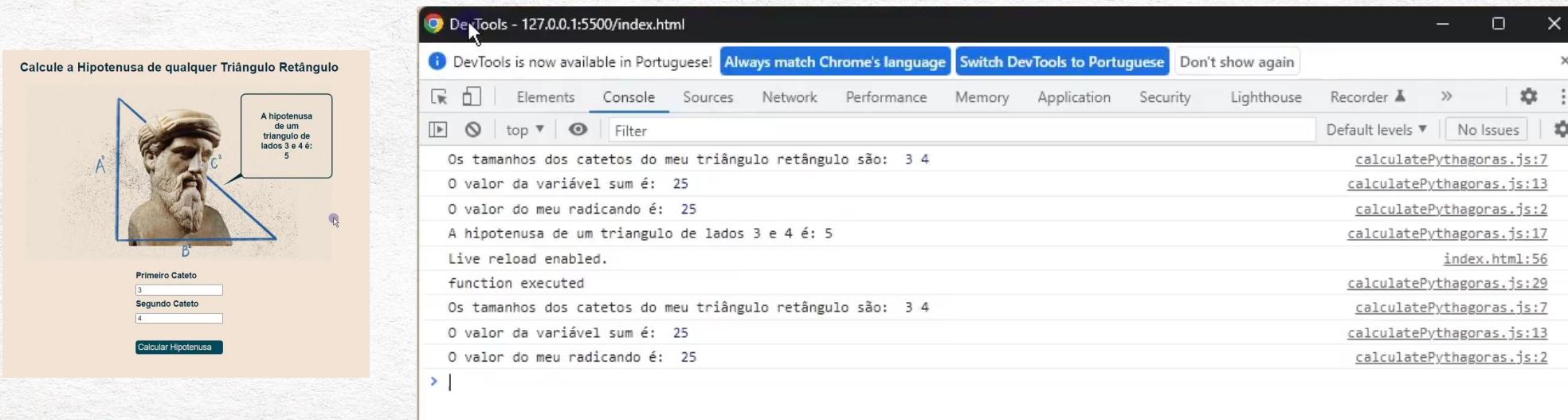
Calcular Hipotenusa

Repare que existem **duas caixas de texto** (Primeiro Cateto e Segundo Cateto) , aplicaremos os valores 3 e 4 , como fizemos via terminal.

A página possui **um botão** - Calcular Hipotenusa, que ao clicarmos **uma mensagem com o resultado** aparece: "A hipotenusa de um triângulo de lados 3 e 4 é: 5"

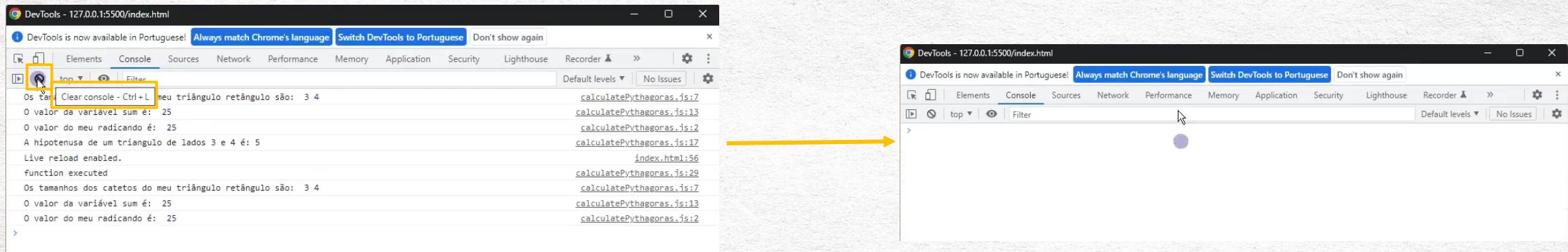
Então conseguimos através de uma interface visual ver o programa funcionando.

Além de ter uma interface visual do nosso programa, podemos utilizar o atalho do nosso navegador F12, para abrir o **DevTools (ferramentas do desenvolvedor)** que todo navegador possui.

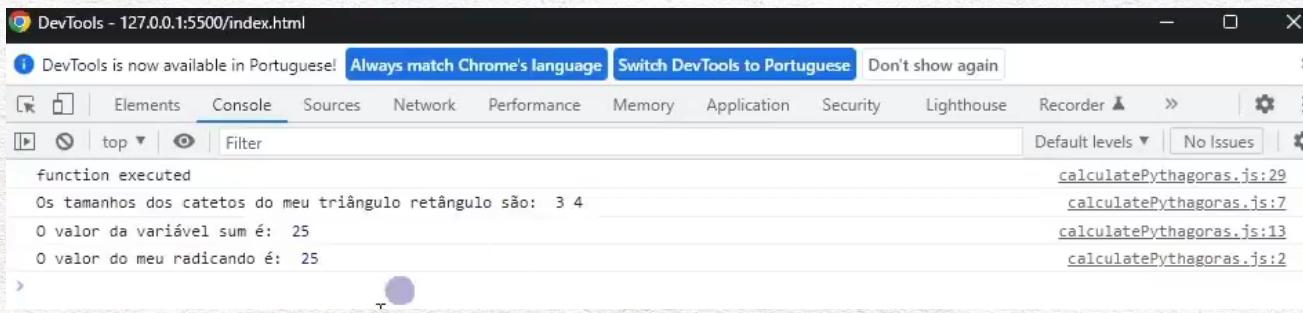


Uma das ferramentas que o **DevTools** possui é a aba **Console**. Ela funcionará de um forma muito parecida com o terminal Node.js que utilizamos no VS Code. Nessa aba, todos aqueles `console.log()` que escrevemos no nosso código, irão aparecer quando executar o programa.

Para limpar o Console iremos clicar no **ícone de limpeza** ou usar o atalho **Ctrl + L** :



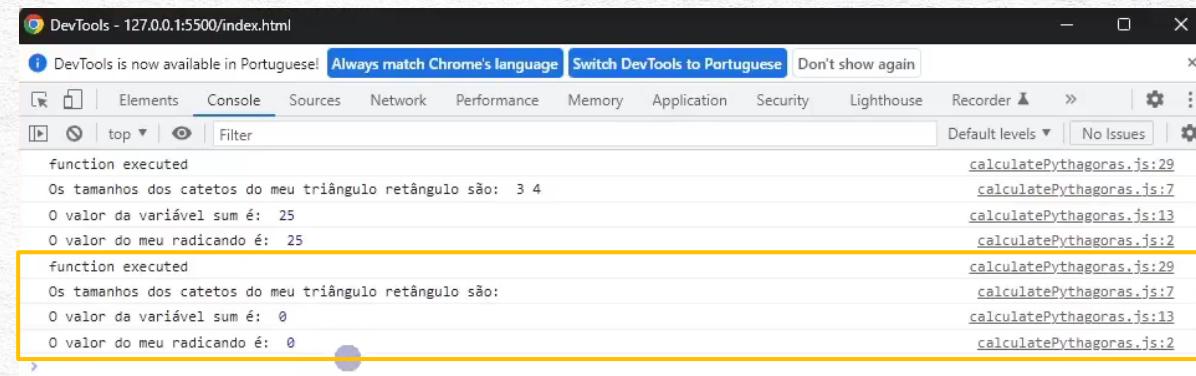
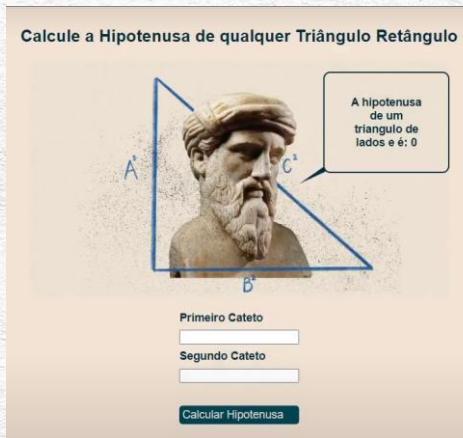
Rodamos novamente nosso programa e conseguimos visualizar os console.log() sendo executados:



Agora além de ver nosso programa funcionando e aprendendo um pouco sobre a **ferramenta DevTools – Console**, vamos colocar em prática o objetivo desse módulo, que é a identificação, leitura e correção de possíveis erros.

Temos o seguinte cenário:

Clicamos no botão Calcular Hipotenusa, sem adicionar nenhum valor aos catetos e tivemos o seguinte retorno de console.



Conseguimos ter a visualização do que está acontecendo, por conta dos nossos `console.log()` estarem retornando, mas se não tivéssemos eles? Afinal, a partir do momento que o programa está sendo utilizado pelo usuário, não é uma boa prática deixar os comandos de console ativos no nosso código, pois todo navegador tem acesso ao DevTools e pode ser utilizado por qualquer usuário para inspecionar a página web.

Para simular a situação citada, vamos comentar todos os `console.log()` do nosso código.



Comentários

Para comentar uma linha de código você deve utilizar `//` ou atalho **Ctrl + /**, veja que a frase ficará de outra cor, indicando que é apenas um comentário e não estará ativo ao executarmos o programa.

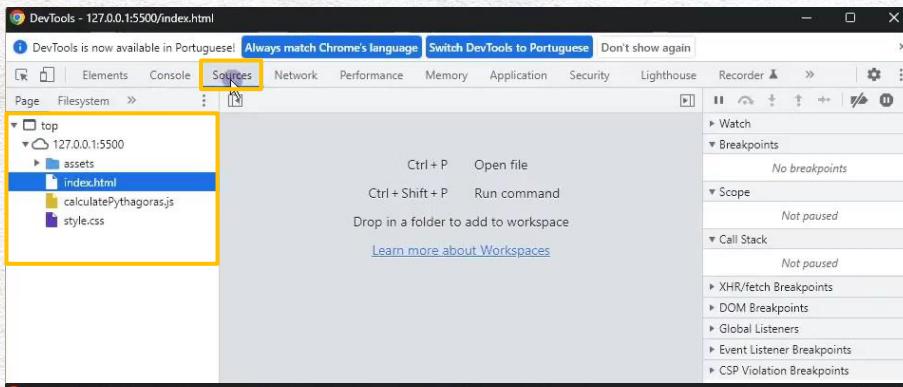
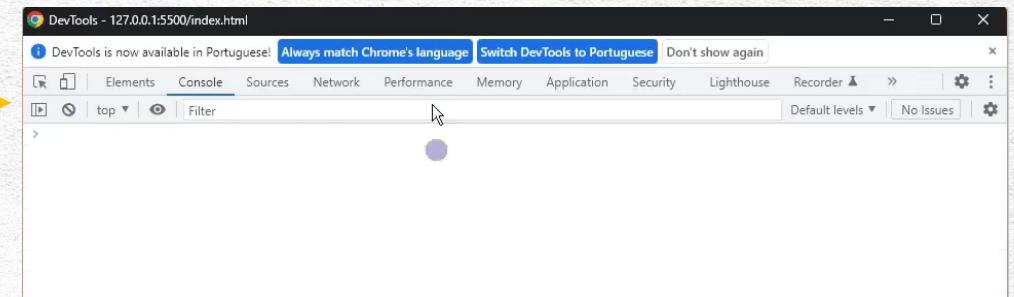
Podemos também comentar em blocos utilizando **/* bloco de código */**.

```
JS calculatePythagoras.js > ...
1  function calculateSquareRoot(radicand) {
2    // console.log('O valor do meu radicando é: ', radicand);
3    return radicand ** (1 / 2);
4  }
5
6  function calculatePythagoras(side1, side2) {
7    // console.log(
8    //   'Os tamanhos dos catetos do meu triângulo retângulo são: ',
9    //   side1,
10   //   side2
11   // );
12  const sum = side1 ** 2 + side2 ** 2;
13  // console.log('O valor da variável sum é: ', sum);
14  return calculateSquareRoot(sum);
15}
16
17 // console.log(
18 //   `A hipotenusa de um triângulo de lados 3 e 4 é: ${calculatePythagoras(3, 4)}`
19 // );
20
21 const firstSideRef = document.getElementById('side-1');
22 const secondSideRef = document.getElementById('side-2');
23 const calculateButton = document.getElementById('calculate-hypotenuse');
24 const answerDiv = document.getElementById('answer');
25
26 function onCalculateButtonClick() {
27   const firstSide = firstSideRef.value;
28   const secondSide = secondSideRef.value;
29   // console.log('function executed');
30   answerDiv.classList.remove('hidden-element');
31   answerDiv.innerHTML = `A hipotenusa de um triângulo de lados ${firstSide} e ${secondSide} é: ${calculatePythagoras(
32     firstSide,
33     secondSide
34   )}
35   .toFixed(3)
36   .replace(/(..)000$/,''))
37 <div class="left-point"></div>`;
38 }
```



Módulo 12 – Dentro do Navegador (11 /19)

Ao comentar / retirar os `console.log()` do nosso código e executar o programa novamente, nada irá ser retornado no DevTools – Console.



```

function calculateSquareRoot(radicand) {
  // console.log('O valor do meu radicando é: ', radicand);
  return radicand ** (1 / 2);
}

function calculatePythagoras(side1, side2) {
  // console.log('Os tamanhos dos catetos do meu triângulo retângulo são: ',
  //             side1,
  //             side2);
  const sum = side1 ** 2 + side2 ** 2;
  // console.log('O valor da variável sum é: ', sum);
  return calculateSquareRoot(sum);
}

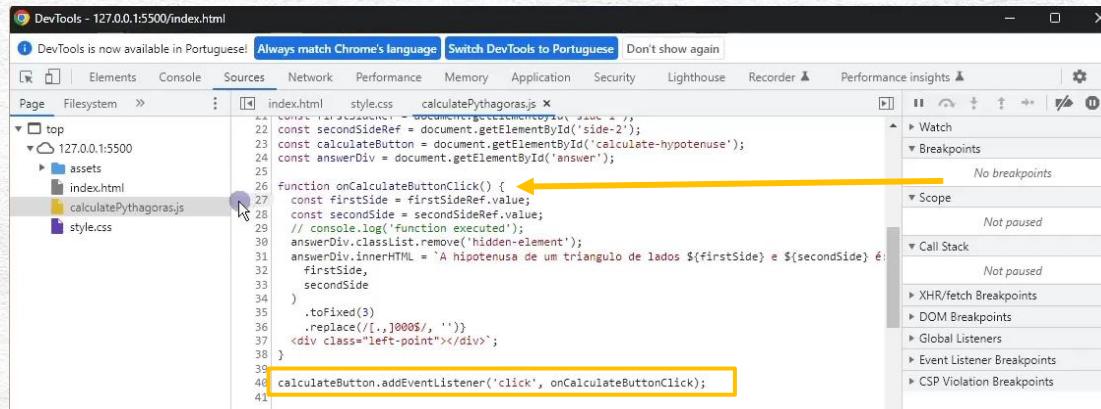
// console.log(
//   'A hipotenusa de um triângulo de lados 3 e 4 é: ${calculatePythagoras(3, 4)}'
// );
  
```

Outra ferramenta que vamos ver do DevTools, é a aba **Source**, que é o local que temos acesso aos arquivos daquele projeto web.

Ao clicarmos em cima do arquivo, ele irá abrir, já que o navegador salva a referência do que foi utilizado para criar a página web.

Módulo 12 – Dentro do Navegador (12 /19)

Sabemos que a última linha do nosso código é a responsável por associar um clique ao elemento do botão Calcular Hipotenusa a ação de executar a função que comanda essa ação.

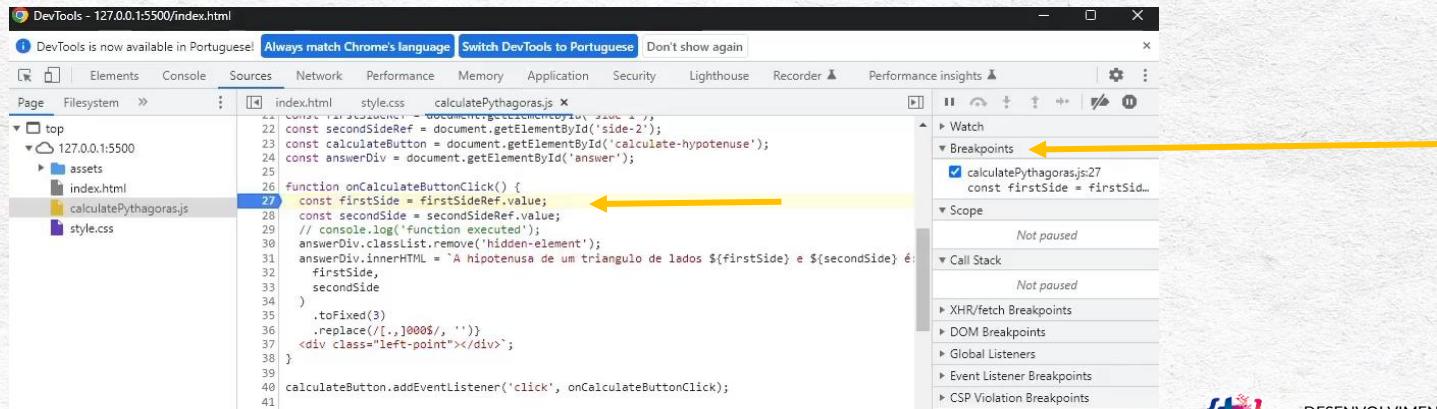


```

DevTools - 127.0.0.1:5500/index.html
  DevTools is now available in Portuguese! Always match Chrome's language Switch DevTools to Portuguese Don't show again
  Elements Console Sources Network Performance Memory Application Security Lighthouse Recorder Performance insights
  Page Filesystem >> index.html style.css calculatePythagoras.js
  top 127.0.0.1:5500 assets index.html calculatePythagoras.js style.css
  1 const firstSideRef = document.getElementById('side-1');
  2 const secondSideRef = document.getElementById('side-2');
  3 const calculateButton = document.getElementById('calculate-hypotenuse');
  4 const answerDiv = document.getElementById('answer');
  5
  6 function onCalculateButtonClick() {
  7   const firstSide = firstSideRef.value;
  8   const secondSide = secondSideRef.value;
  9   // console.log('function executed');
 10   answerDiv.classList.remove('hidden-element');
 11   answerDiv.innerHTML = 'A hipotenusa de um triângulo de lados ${firstSide} e ${secondSide} é:';
 12   firstSide,
 13   secondSide
 14   .toFixed(3)
 15   .replace(/[,]000$/, '')
 16   <div class="left-point"></div>;
 17 }
 18
 19 calculateButton.addEventListener('click', onCalculateButtonClick);
 20
 21
 22
 23
 24
 25
 26
 27
 28
 29
 30
 31
 32
 33
 34
 35
 36
 37
 38
 39
 40
 41

```

Um dos recursos muito interessantes dessa ferramenta são os **BREAKPOINTS**, e para colocar um apenas clicando na linha de código que queremos, nesse caso, o início da nossa função **onCalculateButton**.



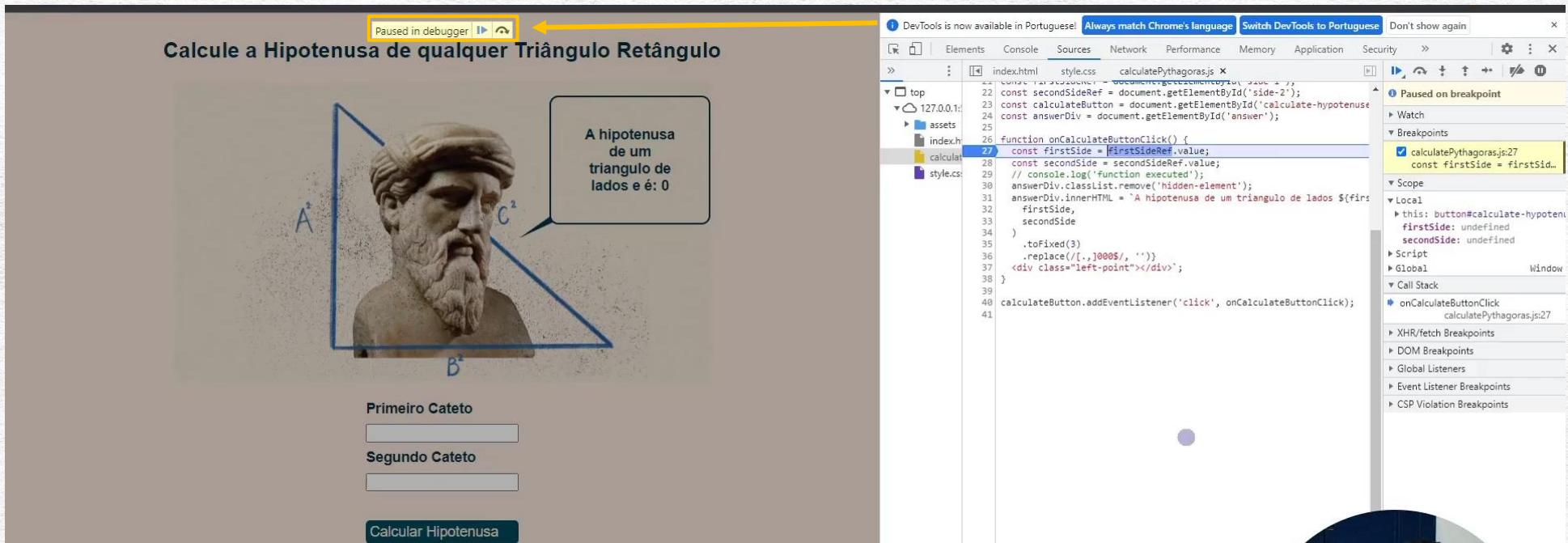
```

DevTools - 127.0.0.1:5500/index.html
  DevTools is now available in Portuguese! Always match Chrome's language Switch DevTools to Portuguese Don't show again
  Elements Console Sources Network Performance Memory Application Security Lighthouse Recorder Performance insights
  Page Filesystem >> index.html style.css calculatePythagoras.js
  top 127.0.0.1:5500 assets index.html calculatePythagoras.js style.css
  1 const firstSideRef = document.getElementById('side-1');
  2 const secondSideRef = document.getElementById('side-2');
  3 const calculateButton = document.getElementById('calculate-hypotenuse');
  4 const answerDiv = document.getElementById('answer');
  5
  6 function onCalculateButtonClick() {
  7   const firstSide = firstSideRef.value;
  8   const secondSide = secondSideRef.value;
  9   // console.log('function executed');
 10   answerDiv.classList.remove('hidden-element');
 11   answerDiv.innerHTML = 'A hipotenusa de um triângulo de lados ${firstSide} e ${secondSide} é:';
 12   firstSide,
 13   secondSide
 14   .toFixed(3)
 15   .replace(/[,]000$/, '')
 16   <div class="left-point"></div>;
 17 }
 18
 19 calculateButton.addEventListener('click', onCalculateButtonClick);
 20
 21
 22
 23
 24
 25
 26
 27
 28
 29
 30
 31
 32
 33
 34
 35
 36
 37
 38
 39
 40
 41

```

Um Breakpoint é um ponto de parada. Toda vez que o navegador ler aquela determinada linha, ele parará a execução e o fluxo será direcionado ao depurador.

Ao executarmos o programa novamente, veja o que ocorre ao utilizarmos um breakpoint:



Ocorrerá uma **parada no local do breakpoint**, e acima da nossa página da web você verá uma caixa : **Paused in debugger**, que é o nosso famoso debugar, retirar possíveis bugs.

Módulo 12 – Dentro do Navegador (14 /19)

Lembra que o JavaScript encapsulou as informações do elementos do arquivo HTML, nessa ferramenta ao passarmos o cursor em cima do código, ela nos mostra qual elemento aquele código pertence, seus valores, propriedades, veja como é uma ferramenta poderosa.

input#side-1 170.34 x 20.17

```

      DevTools is now available in Portuguese. About Feedback Switch DevTools to Portuguese Don't show again
      ▾ Elements ▾ Cons ▾ Network ▾ Timeline ▾ Application ▾ Security ▾ Settings ▾ Help
      ▾ top ▾ 127.0.0.1: ▾ assets ▾ index.html ▾ calculatePythagoras.js ▾ style.css
      1 <input id="side-1" type="text" value="170.34 x 20.17" style="border: 1px solid #ccc; width: 150px; height: 30px; margin-bottom: 10px;"/>
      2 <div id="answer" style="background-color: #f0f0f0; border-radius: 10px; padding: 10px; text-align: center; font-size: 1.2em; font-weight: bold; margin-bottom: 10px;">
      3     A hipotenusa de um triângulo de lados A e B é: 0
      4   
```

 5 <div style="display: flex; justify-content: space-around; align-items: center;">
 6 <input type="text" id="side-1" value="170.34 x 20.17" style="border: 1px solid #ccc; width: 150px; height: 30px; margin-right: 10px;"/>
 7 <input type="text" id="side-2" style="border: 1px solid #ccc; width: 150px; height: 30px; margin-right: 10px;"/>
 8 <button id="calculate-hypotenuse" style="background-color: #0070C0; color: white; border: none; padding: 5px 10px; border-radius: 5px; font-weight: bold; cursor: pointer; margin-left: 10px;"/>
 9 Calcular Hipotenusa
 10

 11

The screenshot shows the Chrome DevTools DevTools interface. The left pane displays the DOM tree with the current file being 'calculatePythagoras.js'. The right pane shows the 'Breakpoints' section where a breakpoint is set on line 27 of the script. A yellow arrow points from the highlighted code in the DOM to the corresponding line in the DevTools script editor. Another yellow arrow points from the highlighted input field in the browser to the variable 'firstSideRef' in the DevTools script editor.

Módulo 12 – Dentro do Navegador (15 /19)

```

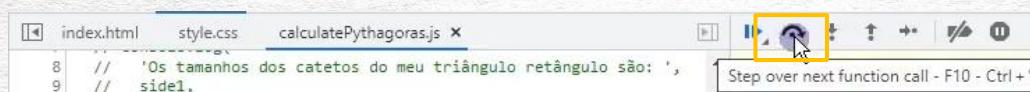
23 const calculateButton = document.getElementById('calculate-hypotenuse');
24 const answerDiv = document.getElementById('answer');
25
26 function onCalculateButtonClick() {
27   const firstSide = firstSideRef.value;
28   const secondSide = secondSideRef.value;
29   // console.log('function executed');
30   answerDiv.classList.remove('hidden-element');
  
```

O valor do elemento é indicado no nosso código pela propriedade **.value**.

Conseguimos verificar passando o cursor nesse exemplo que seu valor é uma **string vazia**.

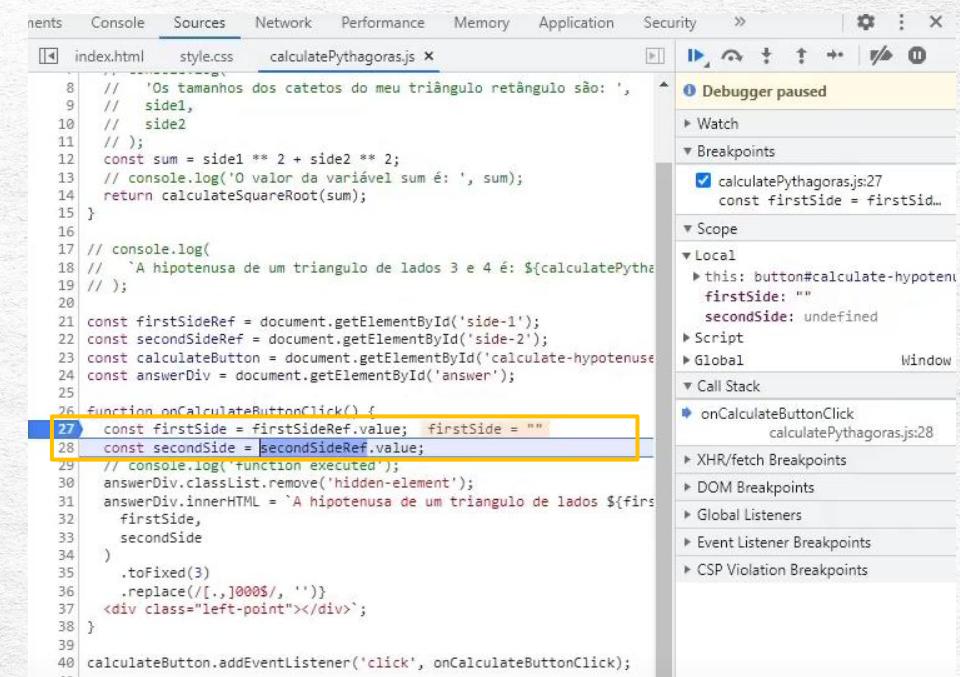
Nesse momento, como não aplicamos nenhum valor a ela e colocamos o breakpoint, ainda não está sendo executado nada.

Utilizando o atalho **F10 - Step over next function**, ou seja, dar um passo para a próxima linha do nosso código.



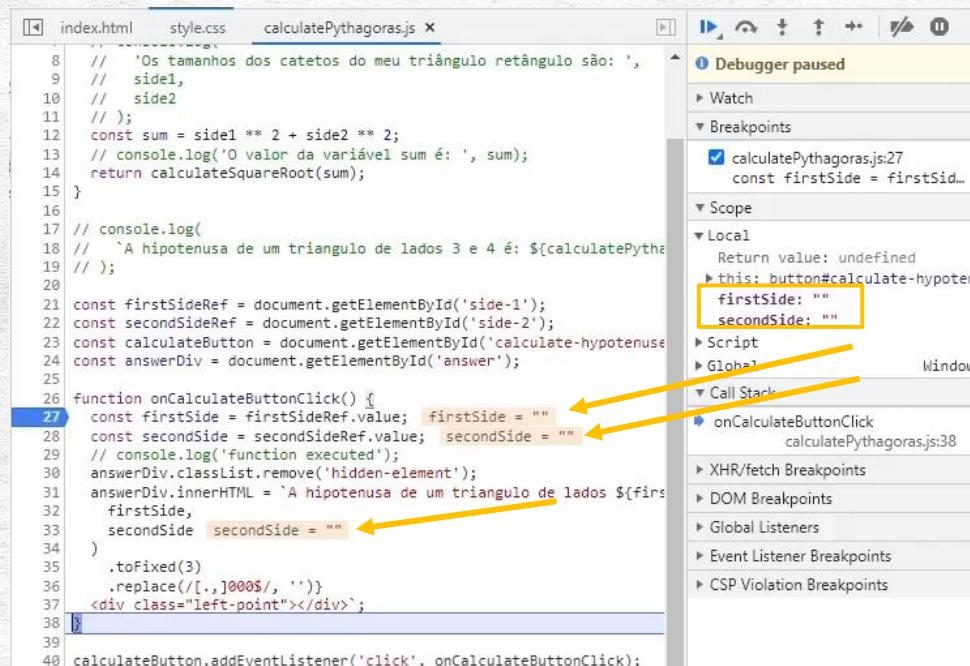
Verificamos na imagem além de passar para a linha de baixo, ele registra as informações do que o nosso código já fez, veja ao lado que **firstSide recebe valor de "" (string vazia)**, e ai ele começa a **executar a próxima linha**.

Isso quer dizer, que conseguimos verificar o nosso código passo a passo, do que ele está atribuindo, retornando, executando.



Módulo 12 – Dentro do Navegador (16 /19)

Continuando a depuração do nosso código, isso quer dizer, executando as demais linhas de código com o nosso debugger. Chegamos a seguinte situação:



The screenshot shows a browser developer tools debugger interface. On the left is the code editor with `calculatePythagoras.js` open. The code defines a function `onCalculateButtonClick` which retrieves values from two input fields and calculates the hypotenuse. On the right is the debugger sidebar with the "Local" scope expanded, showing `firstSide` and `secondSide` both set to an empty string (""). Three yellow arrows point from the code editor to these variables in the sidebar.

```

index.html    style.css    calculatePythagoras.js x
8 // 'Os tamanhos dos catetos do meu triângulo retângulo são: '
9 // side1,
10 // side2
11 //
12 const sum = side1 ** 2 + side2 ** 2;
13 // console.log('O valor da variável sum é: ', sum);
14 return calculateSquareRoot(sum);
15 }
16
17 // console.log(
18 // "A hipotenusa de um triangulo de lados 3 e 4 é: ${calculatePythagoras()}");
19 //
20
21 const firstSideRef = document.getElementById('side-1');
22 const secondSideRef = document.getElementById('side-2');
23 const calculateButton = document.getElementById('calculate-hypotenuse');
24 const answerDiv = document.getElementById('answer');
25
26 function onCalculateButtonClick() {
27   const firstSide = firstSideRef.value; firstSide = ""
28   const secondSide = secondSideRef.value; secondSide = ""
29   // console.log('function executed');
30   answerDiv.classList.remove('hidden-element');
31   answerDiv.innerHTML = `A hipotenusa de um triangulo de lados ${firstSide}, ${secondSide}`.toFixed(3)
32   .replace(/[,]000$/ , '')
33   <div class="left-point"></div>;
34 }
35
36 calculateButton.addEventListener('click', onCalculateButtonClick);
37
38
39
40

```

Identificamos que as nossas variáveis não estão recebendo nenhum valor, ambas estão como strings vazias.

Aqui já seria um indicativo, do porque ao clicarmos no botão o seu retorno vem com o valor de 0.

Mas por questão de aprendizagem vamos continuar os nossos testes.

Então ao atribuir valores aos elementos do HTML e utilizar o breakpoint na mesma função para analisar o passo a passo de sua execução, vamos ver que os valores 3 e 4 são armazenado agora pela variável que armazena o elemento HTML.

Vamos reparar no elemento da linha 31 – **answerDiv.innerHTML**, você não precisa entender-la completamente, o que precisa compreender que existem um campo dentro dessa variável que retorna o nosso texto e retorna o resultado.

The screenshot shows a web application titled "Calcule a Hipotenusa de qualquer Triângulo Retângulo". It features a bust of Pythagoras and a right-angled triangle with sides labeled A², B², and C². A callout box states: "A hipotenusa de um triângulo de lados 3 e 4 é: 5". Below the bust are two input fields: "Primeiro Cateto" with value 3 and "Segundo Cateto" with value 4. A "Calcular Hipotenusa" button is at the bottom. To the right is the Chrome DevTools interface with the "Sources" tab selected, showing the file "calculatePythagoras.js". The code is as follows:

```

1 // DevTools is now available in Portuguese! Always match Chrome's language | Switch DevTools to Portuguese | Don't show again
2
3 // Os tamanhos dos catetos do meu triângulo retângulo são:
4 // side1
5 // side2
6
7 const sum = side1 ** 2 + side2 ** 2;
8 // console.log('O valor da variável sum é: ', sum);
9 return calculateSquareRoot(sum);
10
11 // console.log(
12 //   'A hipotenusa de um triângulo de lados 3 e 4 é: ${calculatePythagoras(3, 4)}');
13
14 const firstSideRef = document.getElementById('side-1');
15 const secondSideRef = document.getElementById('side-2');
16 const calculateButton = document.getElementById('calculate-hypotenuse');
17 const answerDiv = document.getElementById('answer');
18
19 function onCalculateButtonClick() {
20   const firstSide = firstSideRef.value;
21   const secondSide = secondSideRef.value;
22   // console.log('function executed');
23   answerDiv.classList.remove('hidden-element');
24   answerDiv.innerHTML = `A hipotenusa de um triângulo de lados ${firstSide} e ${secondSide} é: ${sum.toFixed(3)}`.replace(/,/g, '.');
25   answerDiv.classList.add('left-point');
26 }
27
28 calculateButton.addEventListener('click', onCalculateButtonClick);
29 
```

A yellow arrow points from the text "A hipotenusa de um triângulo de lados 3 e 4 é: 5" in the browser to the line of code "answerDiv.innerHTML = `A hipotenusa de um triângulo de lados \${firstSide} e \${secondSide} é: \${sum.toFixed(3)}`.replace(/,/g, '.');".

Conseguimos colocar outro breakpoints em nosso código, e através do atalho F8 podemos ser direcionados sempre ao próximo breakpoint existente.

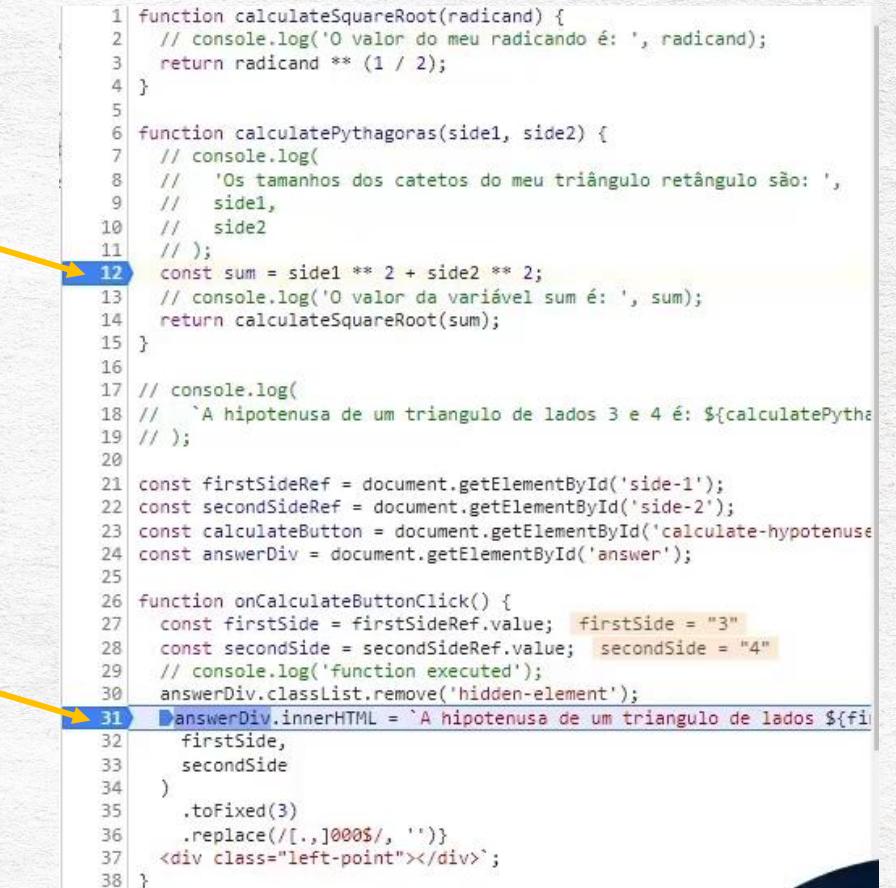
No nosso exemplo iremos retirar o breakpoint do início da nossa função, e vamos adicionar um exatamente na linha do elemento da linha 31 –

answerDiv.innerHTML.

Nesse momento queremos investigar a função que esse elemento está executando, então se apertasse F10, para continuar com o próximo passo, sabemos que ele resolveria o que está nesse breakpoint, mas não me forneceria o passo a passo da função que está dentro dela.

Existem duas formas para analisarmos o cenário acima:

- Colocar um segundo breakpoint dentro da função que está sendo chamada nesse elemento, e ao clicar no F10 ele irá parar nesse passo:



```
1 function calculateSquareRoot(radicand) {  
2     // console.log('O valor do meu radicando é: ', radicand);  
3     return radicand ** (1 / 2);  
4 }  
5  
6 function calculatePythagoras(side1, side2) {  
7     // console.log(  
8     //     'Os tamanhos dos catetos do meu triângulo retângulo são: ',  
9     //     side1,  
10    //     side2  
11    // );  
12    const sum = side1 ** 2 + side2 ** 2;  
13    // console.log('O valor da variável sum é: ', sum);  
14    return calculateSquareRoot(sum);  
15 }  
16  
17 // console.log(  
18 //     'A hipotenusa de um triangulo de lados 3 e 4 é: ${calculatePytha  
19 // }';  
20  
21 const firstSideRef = document.getElementById('side-1');  
22 const secondSideRef = document.getElementById('side-2');  
23 const calculateButton = document.getElementById('calculate-hypotenuse');  
24 const answerDiv = document.getElementById('answer');  
25  
26 function onCalculateButtonClick() {  
27     const firstSide = firstSideRef.value; firstSide = "3"  
28     const secondSide = secondSideRef.value; secondSide = "4"  
29     // console.log('function executed');  
30     answerDiv.classList.remove('hidden-element');  
31     answerDiv.innerHTML = 'A hipotenusa de um triangulo de lados ${fi  
32     firstSide,  
33     secondSide  
34     )  
35     .toFixed(3)  
36     .replace(/[,]000$/ , '')}  
37     <div class="left-point"></div>;  
38 }
```

Módulo 12 – Dentro do Navegador (19 /19)

Ou utilizar o atalho **F11 – Step into next function call**, ou seja, entrar na próxima instrução / função a ser chamada, que neste caso quando ele continuar após o breakpoint, irá entrar na função calculatePythagoras e irá informar o passo a passo da execução dessa função.

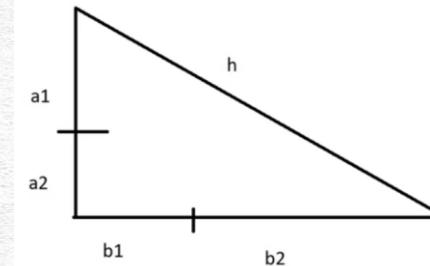
```

1 function calculateSquareRoot(radicand) {
2   // console.log('O valor do meu radicando é: ', radicand);
3   return radicand ** (1 / 2);
4 }
5
6 function calculatePythagoras(side1, side2) { side1 = "5", side2 = "";
7   // console.log(
8   //   'Os tamanhos dos catetos do meu triângulo retângulo são: ',
9   //   side1,
10  //   side2
11  // );
12  const sum = side1 ** 2 + side2 ** 2;
13  // console.log('O valor da variável sum é: ', sum);
14  return calculateSquareRoot(sum);
15 }
16
17 // console.log(
18 //   `A hipotenusa de um triângulo de lados 3 e 4 é: ${calculatePythagoras(3, 4)}`);
19 //
20
21 const firstSideRef = document.getElementById('side-1');
22 const secondSideRef = document.getElementById('side-2');
23 const calculateButton = document.getElementById('calculate-hypotenuse');
24 const answerDiv = document.getElementById('answer');
25
26 function onCalculateButtonClick() {
27   const firstSide = firstSideRef.value;
28   const secondSide = secondSideRef.value;
29   // console.log('function executed');
30   answerDiv.classList.remove('hidden-element');
31   answerDiv.innerHTML = `A hipotenusa de um triângulo de lados ${firstSide}, ${secondSide}`;
32 }
33
34 )

```

Nessa aula aprendemos que mesmo sem os `console.log` o **DevTools é uma ferramenta muito poderosa para o desenvolvedor analisar o seu código**, conseguindo verificar passo a passo a execução do seu programa, o armazenamento de informações e retornos, utilizando o debugger, breakpoints e o cursor do mouse.

Para aprimorarmos nossas habilidades da ferramenta DevTools e análise de erros vamos alterar novamente o nosso código que calcula o teorema de Pitágoras (hipotenusa) seguindo o cenário de um triângulo retângulo do formato ao lado:



Repare que dividimos os catetos em dois valores, ou seja, a soma de $a_1+a_2 = \text{cateto A}$ e $b_1+b_2 = \text{cateto B}$. Para isso precisamos criar essas duas estruturas no nosso código.

Primeiro no arquivo HTML, assim conseguimos ter uma interface visual para essa mudança.

Uma breve explicação sobre o código html, é que cada campo de texto (INPUT) recebe uma etiqueta (LABEL), e a sua comunicação ocorre através do valor atribuído para o ID (identificador) e FOR, eles devem ter os mesmos valores para se relacionar.

The screenshot shows a portion of an HTML file with the following code:

```
<div class="inputs-container">
  <label for="side-1-1">a1</label>
  <input id="side-1-1" type="text" />
  <label for="side-1-2">a2</label>
  <input id="side-1-2" type="text" />
  <label for="side-2-1">b1</label>
  <input id="side-2-1" type="text" />
  <label for="side-2-2">b2</label>
  <input id="side-2-2" type="text" />
  <button id="calculate-hypotenuse">Calcular Hipotenusa</button>
</div>
```

A yellow arrow points from the first two lines of the code (`<label for="side-1-1">a1</label>` and `<input id="side-1-1" type="text" />`) to a screenshot of the user interface below. The user interface shows four input fields labeled a1, a2, b1, and b2, each with a corresponding text input box. Below the inputs is a button labeled "Calcular Hipotenusa".

Agora vamos realizar algumas alterações em nosso código JavaScript, lembrando que nesse momento o objetivo é criarmos um código no qual consiga trabalhar suas habilidades para reconhecer e corrigir erros. Não estamos construindo um código para ele ser eficiente ou enxuto.

```
21 const firstSidePart1Ref = document.getElementById('side-1-1');
22 const firstSidePart2Ref = document.getElementById('side-1-2');
23 const secondSidePart1Ref = document.getElementById('side-2-1');
24 const secondSidePart2Ref = document.getElementById('side-2-2');
```

Em seguida, implementaremos a lógica de somar os lados de A e B, para retornarem o valor de cada lado do cateto corretamente.

A primeira modificação no nosso código é adicionar as referências dos novos elementos que criamos no arquivo HTML.

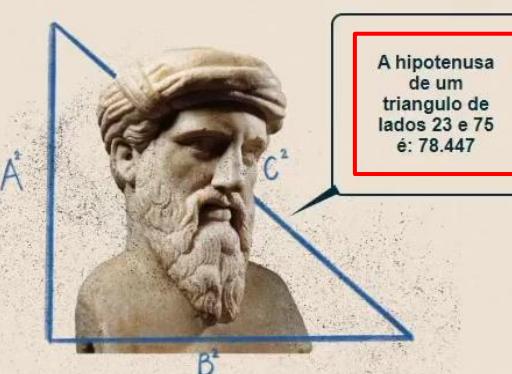
```
42 function add(num1, num2) {
43   const result = num1 + num2;
44   return result;
45 }
```

```
28 function onCalculateButtonClick() {
29   const firstSide = add(firstSidePart1Ref.value, firstSidePart2Ref.value);
30   const secondSide = add(secondSidePart1Ref.value, secondSidePart2Ref.value);
31   // console.log('function executed');
32   answerDiv.classList.remove('hidden-element');
33   answerDiv.innerHTML = `A hipotenusa de um triangulo de lados ${firstSide} e ${secondSide} é: ${calculatePythagoras(
34     firstSide,
35     secondSide
36   )}
37     .toFixed(3)
38     .replace(/[,]000$/ , '')}
39   <div class="left-point"></div>`;
40 }
```

Por último, executamos a nossa função de soma (add) dentro da variável que está armazenando os valores de cada lado do cateto.

Nesse momento iremos utilizar de um triângulo retângulo famoso, que possui lados 5 e 12 e a sua hipotenusa é 13. Então Sabendo que teremos que somar dois valores para gerar um cateto e depois mais dois valores para o segundo cateto, iremos utilizar esse triângulo como base, e verificaremos seu retorno:

Calcule a Hipotenusa de qualquer Triângulo Retângulo



A hipotenusa de um triângulo de lados 23 e 75 é: 78.447

a1

a2

b1

b2

Calcular Hipotenusa

Opa, espera um pouco! Não acabamos de dizer que os valores do cateto A ($a1 = 2$, $a2 = 3$) é o valor 5, e o cateto B ($b1 = 7$, $b2 = 5$) tem valor 12 , e que quando calculado a sua hipotenusa, seu resultado deveria ser 13.

Observe os valores retornados : 23, 75 e 78.447.

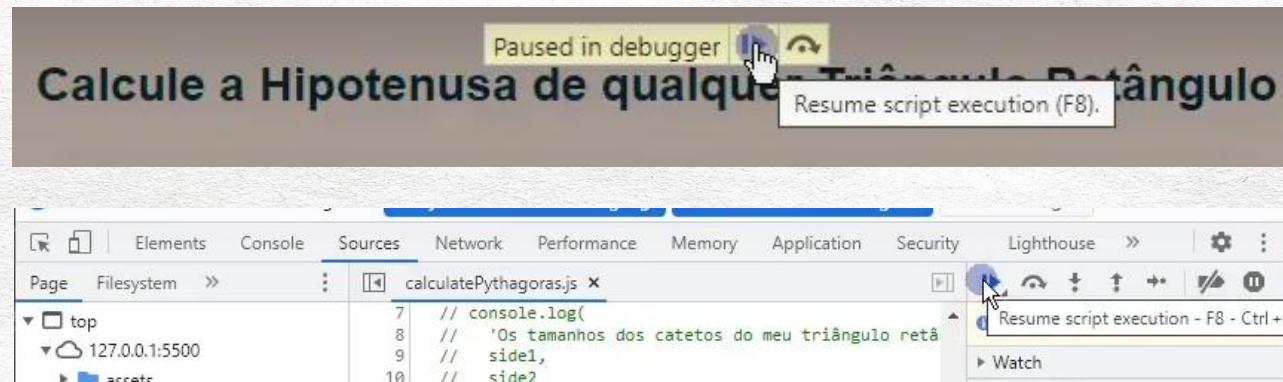
Precisamos analisar o que está acontecendo, afinal sabemos que esse retorno está errado!

Assim como na aula passada, utilizaremos o **DevTools** como ferramenta para explorarmos o nosso código e vamos abrir a aba **Source**, local que está a referência dos arquivos da página web.

Mas antes vamos lembrar que um código JavaScript é lido de cima para baixo, ao colocarmos um **breakpoint**, nosso código iniciará a leitura e ao encontrar esse "ponto de parada" ele irá parar a sua execução, e aguardará até receber alguma instrução, como ir para a próxima linha de instrução (**F10**) ou entrar na execução da próxima instrução/função (**F11**), e assim por diante.

Para adicionarmos um breakpoint, basta clicarmos com o mouse em cima da linha que queremos adicionar e para retirá-lo basta fazer o mesmo procedimento de clicar em cima e pronto.

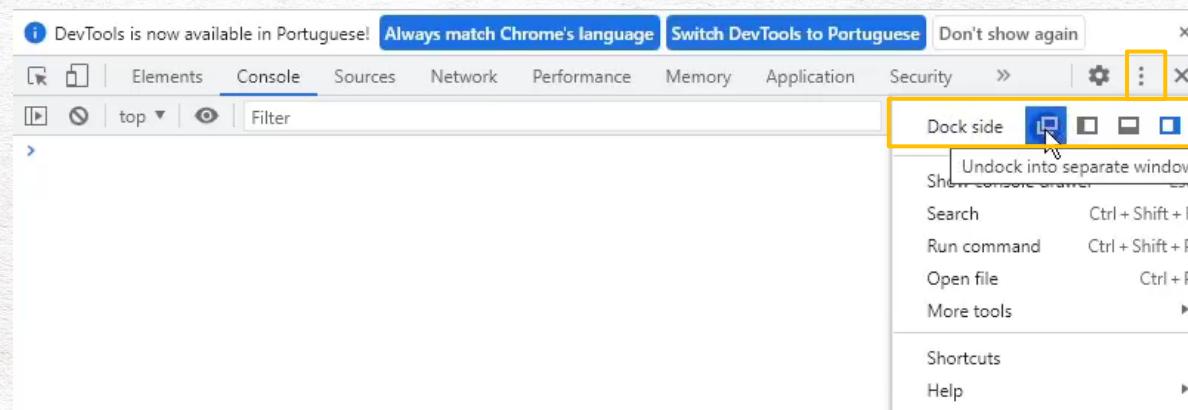
Para 'despausar' o nosso código, ou seja, dar continuidade na sua execução de cima para baixo, podemos clicar nos ícones de play ou utilizar o Atalho **F8**, que ele irá executar o código enquanto não encontrar outro breakpoint ou finalizar o programa:



Dentro do **DevTools** você encontrará o seu **painel de fluxo de controle**, caso prefira não utilizar os atalhos ou não se lembre deles :



E também ao clicar nos três pontinhos, ele abre a **opção** de como você quer a sua **tela do DevTools**, acoplada, solta, na lateral, embaixo e assim por diante. Você pode escolher qual for a melhor opção para você. Recomendamos que estes as maneiras diferentes.



Como falamos o DevTools é uma ferramenta poderosíssima para um desenvolvedor, então vamos conhecer outros campos que ela nos mostra que são muito eficientes na leitura do meu código.

```
// console.log('Os tamanhos dos catetos do meu triângulo retângulo')
// side1,
// side2
const sum = side1 ** 2 + side2 ** 2;
// console.log('O valor da variável sum é: ', sum);
return calculateSquareRoot(sum);

// console.log(
//   `A hipotenusa de um triângulo de lados 3 e 4 é: ${sum}`);
}

const firstSidePart1Ref = document.getElementById('firstSide');
const firstSidePart2Ref = document.getElementById('secondSide');
const secondSidePart1Ref = document.getElementById('firstSide');
const secondSidePart2Ref = document.getElementById('secondSide');
const calculateButton = document.getElementById('calculateButton');
const answerDiv = document.getElementById('answer');

function onCalculateButtonClick() {
  const firstSide = add(firstSidePart1Ref.value, firstSidePart2Ref.value);
  const secondSide = add(secondSidePart1Ref.value, secondSidePart2Ref.value);
  // console.log('Function executed');
  answerDiv.classList.remove('hidden-element');
  answerDiv.innerHTML = `A hipotenusa de um triângulo retângulo é: ${result}`;
}

function add(num1, num2) {
  const result = num1 + num2;
  return result;
}

calculateButton.addEventListener('click', onCalculateButtonClick);
```

Na lateral do DevTools aparece para nós algumas informações, muito úteis para analisarmos.

A primeira aba é dos **BREAKPOINTS**, clicando nela irá aparecer a lista de todos os breakpoints do nosso código e local onde estão.

A próxima aba é **SCOPE**, lugar onde fica armazenada as nossas variáveis locais e globais.

A aba **Call Stack**, que está logo abaixo, é onde a nossa **pilha de execução** será mostrada, ou seja, a ordem que as funções chamam outras funções. Ele executa a primeira função chamada, se dentro dela houver uma outra função ele irá executar essa nova função até o final ou até encontrar um outra função dentro dela.

Conseguimos entender melhor com o seguinte exemplo :

A **função calculatePythagoras** começa a ser executada, quando encontramos a **função calculateSquareRoot**, o nosso programa executa ela até o final e depois retorna com resultado para a **função calculatePythagoras**, que continuará sua execução.

The screenshot shows a code editor with the following JavaScript code:

```
41
42 function add(num1, num2) { num1 = "7", num2 = "5"
43   const result = num1 + num2;
44   return result;
```

Below the code, a scope inspector is open. On the left, under the 'Local' scope, 'num1' is shown with the value '7' and 'num2' with the value '5'. On the right, under the 'Local' scope, 'num1' is shown with the value '7' and 'num2' with the value '5'. A yellow arrow points from the 'num1' entry in the left scope to its corresponding entry in the right scope.

E olha que bacana, através da aba de **SCOPE**, além de **observar** quais **variáveis** o nosso código está armazenando, se clicarmos no valor delas conseguimos ter o poder de **altera-las**.

Nesse caso em específico estamos suspeitando que o tipo do valor da variável está estranha, então para verificar iremos testar os valores que esperamos que ela receba com o tipo certo, ao invés de "7" e "5", testaremos 7 e 5.

Mas isso não é a mesma coisa?? **Não!**

Lembre-se que para realizar operações Aritméticas são necessários valores numéricos, e quando os valores estão entre aspas identificamos pelo tipo String, ou seja, um texto, e não conseguimos realizar operações matemáticas com textos.

Então acabamos de analisar que os textos estão fazendo uma concatenação, por isso o resultado "75" e não 12.

Como a análise, percebemos que a lógica da conta nunca esteve errada, mas sim quando o JavaScript captura a primeiro momento as variáveis assumindo que elas são textos, apenas em casos que são muito explícitos o tipo da variável, o JavaScript sempre terá em primeiro momento essa leitura.

Por exemplo, quando elevamos um número ao quadrado, o JavaScript sabe que não podemos levar um texto ao quadrado, mas em relação ao **operador +**, ele não consegue distinguir se será uma operação matemática ou uma concatenação entre textos.

Esse é um caso no qual você programador, sempre terá que ter o cuidado de converter os seus textos em números, quando assim for necessário.

Para isso utilizamos uma função do JavaScript, o **parseInt()** que analisa um argumento string e retorna um inteiro. Então é passado o valor que quer modificar entre os seu parênteses, e dessa forma a conversão será aplicada.

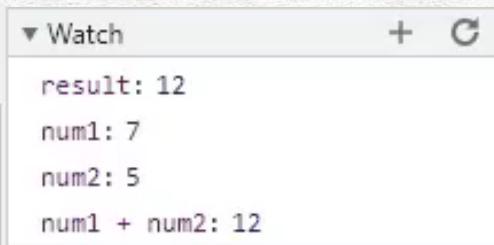
```
const firstSide = add(parseInt(firstSidePart1Ref.value), parseInt(firstSidePart2Ref.value));
const secondSide = add(parseInt(secondSidePart1Ref.value), parseInt(secondSidePart2Ref.value));
```

Além de testar valores de variáveis o DevTools também permite você escrever nos arquivos da página, mas isso não quer dizer que você estará efetivamente alterando a página de Web, quando você atualizar o seu navegador, ele retornará da maneira que estava originalmente.

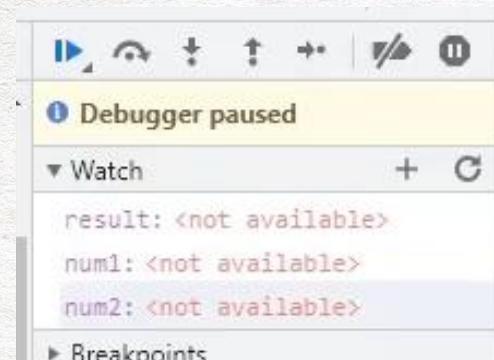
Porque imagine uma ferramenta tão poderosa, na qual qualquer pessoa teria acesso em modificar e alterar sites, então só para deixar claro, **essas alterações são apenas testes** e "brincadeiras" que fazemos para testar possibilidades e aprender, mas a partir do momento que você **atualizar sua página** ela voltará o **código original**.

Existe mais uma ferramenta muito bacana do DevTools, que é a aba **Watch** que significa observar, ela permite inserir um dado que queremos rastrear / observar.

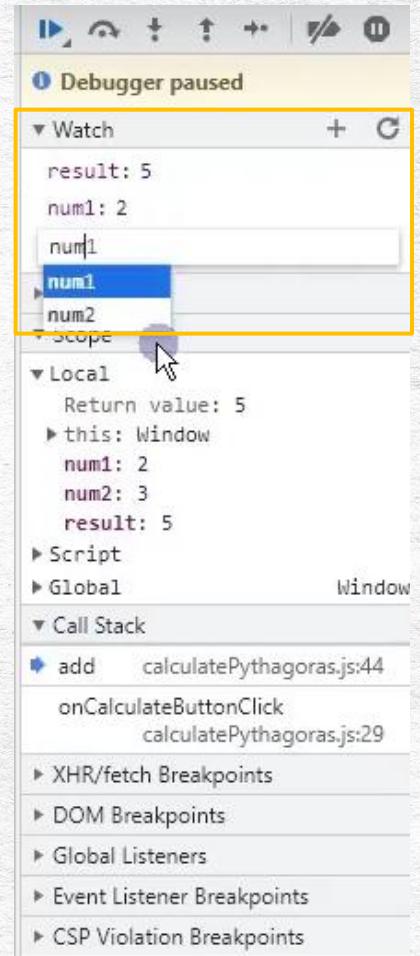
Você clica no sinal de + e digita um valor que quer observar, no caso ao lado estamos observando as **variáveis result, num1** por exemplo.



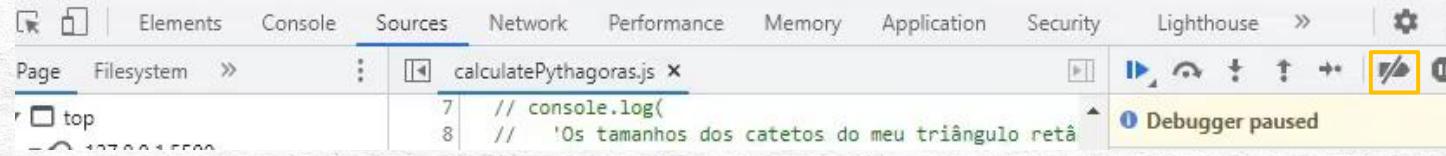
A ferramenta aceita também valores compostos, rastreio de operações matemáticas, não apenas elementos simples.



E enquanto variável não receber um valor para ser observado, aparecerá como **not available**, ou seja, não disponível naquele momento, pois o código ainda não executou a atribuição dela ainda.

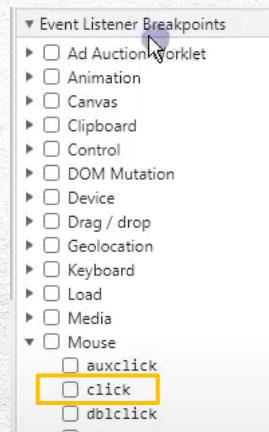


Para desativar ou ativar o breakpoints que colocamos no nosso código, dentro do controle de fluxo existe um botão que ativa e desativa eles, ou o atalho CTRL + F8.



Para finalizarmos, vamos a uma breve explicação sobre o Botão do nosso programa.

O nosso código associa a lógica do botão a um **EVENTO**, que nesse caso é o evento de click, e no DevTools, também existe a aba **Event Listener Breakpoints** que conseguimos criar break points nesses eventos.



E os eventos são elementos muito importantes em páginas de Web, e ao longo do curso iremos ter mais contato com esse conceito. Então, guarda esse nome Evento, pois iremos utilizá-lo mais à frente e é um conceito muito importante para entendermos.

```
JS calculatePythagoras.js > ...
1  function calculateSquareRoot(radicand) {
2  |  return radicand ** (1 / 2);
3  }
4
5  function calculatePythagoras(side1, side2) {
6  |  const sum = side1 ** 2 + side2 ** 2;
7  |  return calculateSquareRoot(sum);
8  }
9
10 console.log(process.argv); ←
11
12 console.log(
13 | `A hipotenusa de um triangulo de lados 3 e 4 é: ${calculatePythagoras(3, 4)}`
14 );
15
```

Propositalmente o código que iremos utilizar nesse momento, estará menor do que o que vimos anteriormente. Vamos retornar para um código mais simples e os testes e análises serão feitos através do VS Code e terminal.

Para isso foi retirado a parte da nossa interface visual (HMTL e CSS) e você percebe que já tem um pouco mais de familiaridade com esse código.

Mas repare que adicionamento uma informação dentro do `console.log`, acrescentamos o **process.argv**.

Vamos executar o nosso programa para verificar o que ele imprimirá na tela.

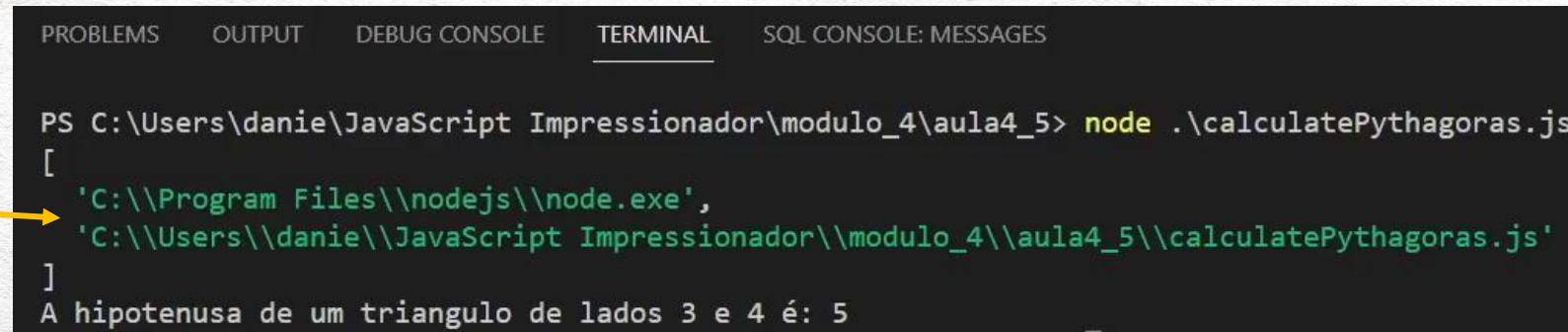
Antes de analisar o retorno do programa, você precisa entender o conceito de Array.

Um **ARRAY** é um conjunto de dados e são utilizados para armazenar mais de um valor em uma única variável.

Para criarmos uma Array, iremos declarar uma variável e o valor atribuído à ela será colocado entre **colchetes []**.

declaração do tipo + nome da variável + sinal de igual (=) + [informação1, informação2, informação3...]

Vamos observar nosso programa e o que retornou para o nosso **console.log(process.argv)** um **array** com duas informações.



The screenshot shows the VS Code interface with the Terminal tab selected. The terminal window displays the following text:

```
PS C:\Users\danie\JavaScript Impressionador\modulo_4\aula4_5> node .\calculatePythagoras.js
[
  'C:\\Program Files\\nodejs\\node.exe',
  'C:\\Users\\danie\\JavaScript Impressionador\\modulo_4\\aula4_5\\calculatePythagoras.js'
]
A hipotenusa de um triangulo de lados 3 e 4 é: 5
```

A yellow arrow points to the first element of the array, which is the path to the Node.js executable.

Até então para executarmos o programa, utilizando o comando via terminal **node ./arquivo.js**, mas não sei se você já teve a curiosidade de adicionar outras informações após esse comando. Então vamos realizar esse teste agora:

```
PS C:\Users\danie\JavaScript Impressionador\modulo_4\aula4_5> node .\calculatePythagoras.js 2312451251
[
  'C:\\Program Files\\nodejs\\node.exe',
  'C:\\Users\\danie\\JavaScript Impressionador\\modulo_4\\aula4_5\\calculatePythagoras.js',
  '2312451251'
]
A hipotenusa de um triangulo de lados 3 e 4 é: 5
```

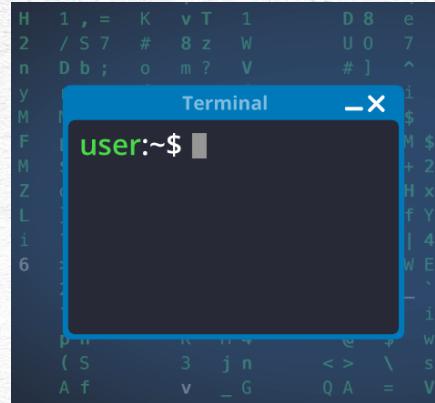
Repare que essa nova informação que passamos no nosso terminal, foi adicionada ao array e impressa via **console.log(process.argv)**.

Vamos fazer mais um teste e agora você irá observar que ao separarmos por um espaço, ele já entende que é uma nova informação, retornando um array com 4 informações agora.

```
PS C:\Users\danie\JavaScript Impressionador\modulo_4\aula4_5> node .\calculatePythagoras.js 23124 51251
[
  'C:\\Program Files\\nodejs\\node.exe',
  'C:\\Users\\danie\\JavaScript Impressionador\\modulo_4\\aula4_5\\calculatePythagoras.js',
  '23124',
  '51251'
]
A hipotenusa de um triangulo de lados 3 e 4 é: 5
```

O que estamos fazendo é algo muito interessante, mas afinal o que está acontecendo?

O **argv** é um array de strings e cada string desse array é um parâmetro da linha de comando, ou seja, ele dá ao programador acesso à linha de comando com o qual o programa foi chamado. E por isso o seu retorno inicial é os caminhos do node e do nosso programa.



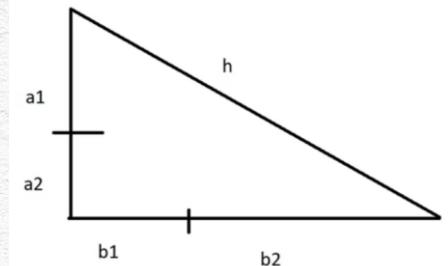
Podemos então pensar que se temos "alguém" dentro do código que tem acesso as informações que passo via terminal, no momento de execução do meu código, isso quer dizer que podemos adaptar o nosso código para que ele reaja a essas informações. Consequentemente conseguimos deixar nosso código muito mais inteligente.

Então vamos colocar a mão na massa e utilizar desse recurso para deixar nosso código mais dinâmico. Ao invés de colocarmos os valores de forma estática, construiremos o nosso programa para interagir com os valores passados pelo terminal, ou seja, os dois valores números que precisamos para calcular a Hipotenusa no nosso código, serão passados pelo nosso terminal.

Lembrando que algumas alterações do nosso código irão servir para o nosso objetivo principal do módulo, que são os erros e aprender técnicas para descobrir e resolver eles.

Primeiro adicionaremos uma função de soma no nosso código, para replicar o mesmo exemplo que utilizamos no navegador.

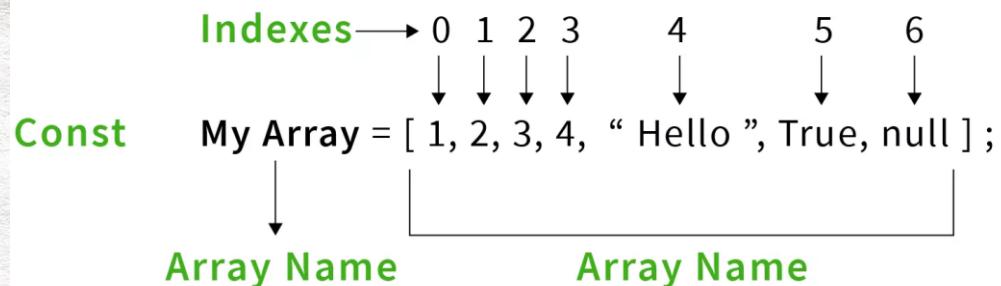
```
10  function add(part1, part2){  
11    return part1 + part2;  
12 }
```



Nesse cenário esperamos que seja passado 4 valores no nosso terminal:

```
PS C:\Users\danie\JavaScript Impressionador\modulo_4\aula4_5> node .\calculatePythagoras.js 2 3 1 2  
[  
  'C:\\Program Files\\nodejs\\node.exe', X  
  'C:\\Users\\danie\\JavaScript Impressionador\\modulo_4\\aula4_5\\calculatePythagoras.js', X  
  '2',  
  '3',  
  '1',  
  '2'  
]  
A hipotenusa de um triangulo de lados 3 e 4 é: 5
```

Entendemos que os 2 primeiros argumentos serão as informações do node e do programa, então a primeiro momento devemos ignorá-los, afinal as informações que queremos capturar vem depois deles.



Quando queremos acessar uma informação de um **Array** utilizaremos o termo índice, que são as "casas" que cada informação está armazenada. O primeiro valor sempre terá **índice** no valor de **0**. E dentro desse array pode ser armazenado valores de todos os tipos.

Criaremos uma variável para armazenar as entradas do terminal e ela será um subarray, ou seja, um array menor (variável nova) gerado por um array maior(process.argv).

E para capturarmos as informações corretas, utilizaremos um método que todo array carrega chamado `splice()`.

*Não se preocupe nesse momento em entender o que é um método, pois ao longo do curso iremos estar aprendendo sobre esse conceito

O `splice()` fará um corte no array, a partir do índice que passarmos como parâmetro e retorna um subarray. No nosso caso queremos capturar as informações a partir do índice 2, então faremos um corte nesse ponto e retornaremos as informações seguintes do nosso array.

```
16 const userInputs = process.argv.splice(2);
```

```
PS C:\Users\danie\JavaScript Impressionador\modulo_4\aula4_5> node .\calculatePythagoras.js 2 3 1 2
[
  'C:\\Program Files\\nodejs\\node.exe', 0
  'C:\\Users\\danie\\JavaScript Impressionador\\modulo_4\\aula4_5\\calculatePythagoras.js', 1
  '2', 2
  '3', 3
  '1', 4
  '2', 5
]
A hipotenusa de um triangulo de lados 3 e 4 é: 5
```

As informações que queremos capturar vão do índice 2 ao 5.

Após esse tratamento inicial de armazenar as informações das entradas em uma variável, podemos entender o que ocorre imprimindo na tela que a **variável userInputs** retorna exatamente os valores que queremos.

```
PS C:\Users\danie\JavaScript Impressionador\modulo_4\aula4_5> node .\calculatePythagoras.js 2 3 1 2
[
  'C:\\Program Files\\nodejs\\node.exe',
  'C:\\Users\\danie\\JavaScript Impressionador\\modulo_4\\aula4_5\\calculatePythagoras.js',
  '2',
  '3',
  '1',
  '2'
]
[ '2', '3', '1', '2' ] ← SUBARRAY
A hipotenusa de um triangulo de lados 3 e 4 é: 5
```

Continuando a construção do nosso código, criaremos os lados do nosso triângulo, para isso aplicaremos a seguinte lógica:

```
16 const side1 = add(userInputs[0], userInputs[1]);  
17 const side2 = add(userInputs[2], userInputs[3]);
```

Repare, para capturar a informação do array, precisamos chamar a **variável** que o array está armazenado e entre **colchetes []**, passaremos o **índice** do valor que queremos.

Nesse caso, quando criamos o nosso subarray o primeiro valor será armazenado no índice 0, **SEMPRE o primeiro índice será 0**, mesmo que seja um subarray, ou seja, toda vez que criarmos um novo array ou subarray o índice inicial terá valor 0.



Para finalizar nosso código iremos alterar nosso retorno de console.log().

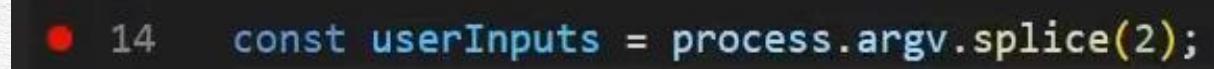
```
19 console.log(  
20 | `A hipotenusa de um triangulo de lados ${side1} e ${side2} é: ${calculatePythagoras(side1, side2)}`  
21 );
```

Ao executarmos nosso programa observamos que replicamos o mesmo erro que ocorreu quando utilizamos o navegador e o DevTools, mas vamos imaginar que nós ainda não sabemos o que está acontecendo.

```
PS C:\Users\danie\JavaScript Impressionador\modulo_4\aula4_5> node .\calculatePythagoras.js 2 3 1 2
A hipotenusa de um triangulo de lados 23 e 12 é: 25.942243542145693
```

Assim como navegador, o **VS Code** também possui uma ferramenta para criar breakpoints, executar o debugger, analisar as variáveis sendo alteradas e assim por diante.

Para criarmos um **breakpoint**, assim como no DevTools, você deve clicar na linha que quer inserir a parada até aparecer um bolinha vermelha:



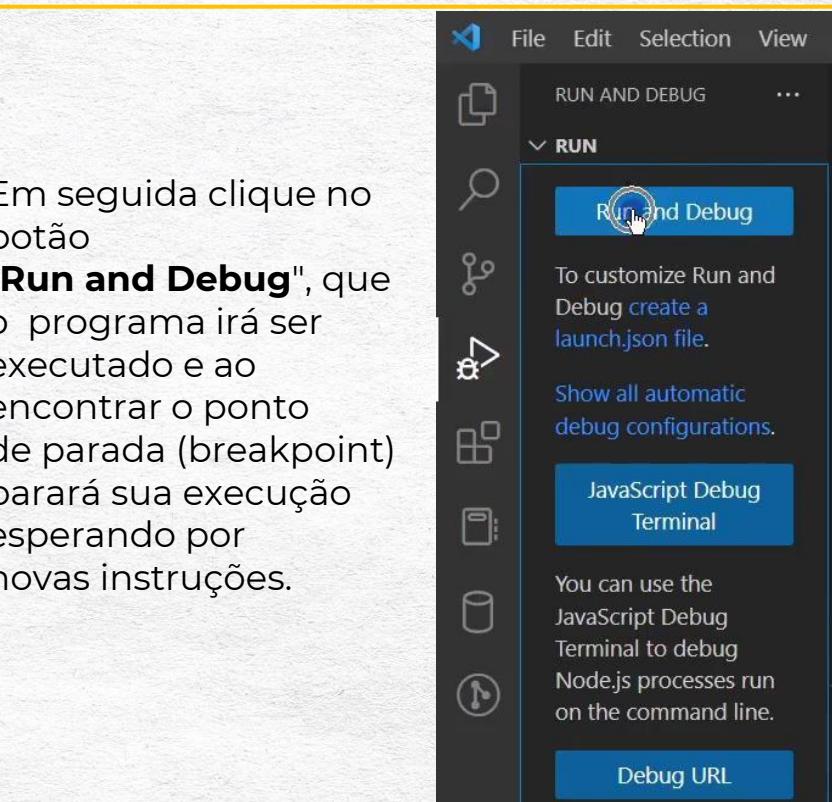
```
● 14 const userInputs = process.argv.splice(2);
```

Para executarmos o Debugger:



Clique no ícone do **Run and Debug**.

Ao iniciar a execução do **Run and Debug**, o controle de fluxo estará disponível na tela para você executar as próximas instruções.



Em seguida clique no botão **"Run and Debug"**, que o programa irá ser executado e ao encontrar o ponto de parada (breakpoint) parará sua execução esperando por novas instruções.

Na lateral esquerda do VS Code é o local das ferramentas de análise de Variáveis, Call Stack, Breakpoints, Watch e elas terão as mesmas funções quando utilizamos o DevTools no navegador.

A screenshot of the Visual Studio Code interface. On the left, there's a sidebar with several sections: 'VARIABLES' (highlighted with a yellow arrow), 'WATCH' (highlighted with a yellow arrow), 'CALL STACK' (highlighted with a yellow arrow), 'LOADED SCRIPTS' (highlighted with a yellow arrow), and 'BREAKPOINTS' (highlighted with a yellow arrow). The main area shows a JavaScript file named 'calculatePythagoras.js'. The code defines a function to calculate the hypotenuse of a triangle given two sides. The 'WATCH' section shows 'side1' and 'side2' both set to 'undefined'. The 'CALL STACK' section shows the current stack frame. The 'LOADED SCRIPTS' section lists the module being run. The 'BREAKPOINTS' section shows one breakpoint at line 14. The status bar at the bottom indicates the command 'PS C:\Users\danie\JavaScript Impressionador\modulo_4\aula4_5> node .\calculatePythagoras.js 2 3 1 2' and the output 'A hipotenusa de um triangulo de lados \${{side1}} e \${{side2}} é: \${{calculatePythagoras(side1, side2)}}'.

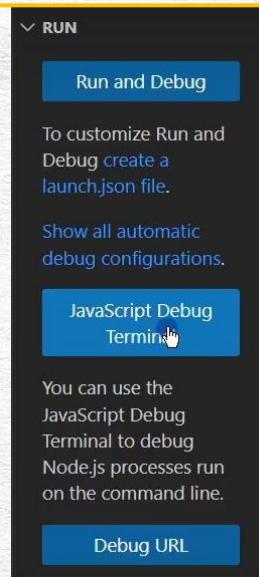
A screenshot of the 'VARIABLES' panel in VS Code. It shows the 'Local' scope where 'side1' and 'side2' are both listed as 'undefined'. There's also a 'Global' scope entry for 'userInputs'. The 'WATCH' section at the bottom shows the expression 'side1 + side2' with the value 'NaN'. The sidebar on the left has sections for 'VARIABLES', 'WATCH', 'CALL STACK', 'LOADED SCRIPTS', and 'BREAKPOINTS'.

No Vs Code, como não atribuímos ainda nenhum valor as variáveis, ele retorna o valor especial NaN, not a number.

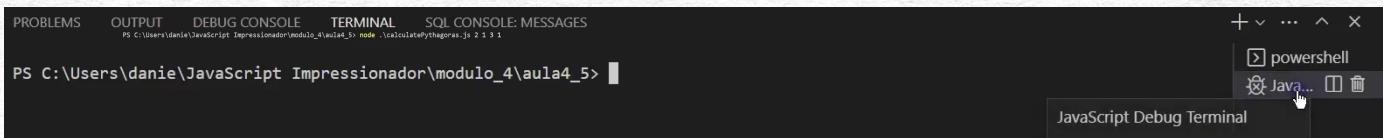
O debugger no VS Code, servirá para você mapear seus códigos que não necessariamente sejam página de web, e sim programa que são executados no nosso terminal via node.

E pensando nisso, você reparou que executamos o Debugger clicando no botão "Run and Debug", isso faz com que ele comece a execução do nosso código, mas como ele irá analisar os valores se o nosso programa recebe as informações via terminal?

Para isso devemos utilizar a segunda opção do Debugger do VS Code, "Javascript Debug Terminal":



Após clicar, ele irá abrir um outro terminal vinculado com o debugger ao nosso terminal.



E executamos o nosso programa:

```
PS C:\Users\danie\JavaScript Impressionador\modulo_4\aula4_5> node .\calculatePythagoras.js 2 1 3 1
```

Módulo 12 – DevTools no VS Code (13 /15)

Dessa forma nosso programa será executado via terminal vinculado com o Debugger , para isso a informação Debugger attached, será impressa no terminal.

The screenshot shows the VS Code interface with the following details:

- Editor:** The file `calculatePythagoras.js` is open, displaying the following code:

```
9
10    function add(part1, part2) {
11        return part1 + part2;
12    }
13
14    const userInputs = process.argv.splice(2);
15
16    const side1 = add(userInputs[0], userInputs[1]);
17    const side2 = add(userInputs[2], userInputs[3]);
18
19    console.log(
20        `A hipotenusa de um triangulo de lados ${side1} e ${side2} é: ${calculatePythagoras(
21            side1,
22            side2
23        )}`);
24
```
- Variables View:** Shows the local variables state, including `_dirname`, `_filename`, `add`, `calculatePythagoras`, `calculateSquareRoot`, `exports`, `module`, `require`, `side1`, `side2`, `this`, and `userInputs`. `userInputs` is listed as `undefined`.
- Watch View:** Shows the expression `side1 + side2` with the value `Nan`.
- Terminal:** Displays the command `PS C:\Users\danie\JavaScript Impressionador\modulo_4\aula4_5> node .\calculatePythagoras.js 2 1 3 1` and the message `Debugger attached.`

E da mesma forma que conseguimos analisar o que estava ocorrendo no código no navegador e no código Javascript, agora podemos analisar vinculando as informações do terminal.

Ao executarmos o nosso programa o retorno através do Debugger será o mesmo que no navegador, afinal ainda não corrigimos o erro que nós já sabemos que existe:

```

VS CODE SCREENSHOT: calculatePythagoras.js
Variables (Local):
  side1: '21'
  side2: '31'

Watch:
  side1 + side2: '2131'

Code Editor (calculatePythagoras.js):
  9
  10 function add(part1, part2) {
  11   return part1 + part2;
  12 }
  13
  14 const userInputs = process.argv.splice(2);
  15   '21' (highlighted by arrow)
  16 const side1 = add(userInputs[0], userInputs[1]);
  17 const side2 = add(userInputs[2], userInputs[3]);
  18
  19 console.log(
  20   `A hipotenusa de um triangulo de lados ${side1} e ${side2} é: ${calculatePythagoras(
  21     side1,
  22     side2
  23   )}`);
  .
  .
  .
  PS C:\Users\danie\JavaScript Impressionador\modulo_4\aula4_5> node .\calculatePythagoras.js 2 1 3 1
  Debugger attached.
  
```

Mas o bacana do VS Code é que ele é mais intuitivo devido ao estilo que usa na construção do nosso código, veja que ao testarmos o nosso código, a coloração dos tipos de variáveis muda, nesse caso repare que nossas variáveis tem uma cor laranja, simbolizando um texto (string).

Quando alteramos para um número (number) a cor dela ficará amarelo/verde, o que já facilita na compreensão de onde o erro está.

```

VS CODE SCREENSHOT: calculatePythagoras.js (fixed)
Variables (Local):
  side1: 21
  side2: '31'

Code Editor (calculatePythagoras.js):
  9
  10 function add(part1, part2) {
  11   return part1 + part2;
  12 }
  13
  14 const userInputs = process.argv.splice(2);
  15   21 (highlighted by arrow)
  16 const side1 = add(userInputs[0], userInputs[1]);
  17 const side2 = add(userInputs[2], userInputs[3]);
  18
  19 console.log(
  20   `A hipotenusa de um triangulo de lados ${side1} e ${side2} é: ${calculatePythagoras(
  21     side1,
  22     side2
  23   )}`);
  .
  .
  .
  PS C:\Users\danie\JavaScript Impressionador\modulo_4\aula4_5> node .\calculatePythagoras.js 2 1 3 1
  Debugger attached.
  
```

Corrigiremos o erro, transformando as strings em números através de um outro método chamado **Number()**.

```
const side1 = add(Number(userInputs[0]), Number(userInputs[1]));
const side2 = add(Number(userInputs[2]), Number(userInputs[3]));
```

Só executarmos o nosso programa novamente que tudo funcionará de maneira correta.

```
PS C:\Users\danie\JavaScript Impressionador\modulo_4\aula4_5> node .\calculatePythagoras.js 2 1 3 1
Debugger attached.
A hipotenusa de um triangulo de lados 3 e 4 é: 5
```

Resumindo, além da ferramenta DevTools, o VS Code também nos fornece essa ferramenta poderosa que é o Debugger, e com ela conseguimos mapear tudo o que está ocorrendo dentro do nosso programa, facilitando a análise e correção de possíveis erros do nosso código.

Módulo 13

TRATAMENTO DE ERRO

TRATAMENTO DE ERRO

TRATAMENTO DE ERRO



PROGRAMAÇÃO DEFENSIVA

A programação defensiva em JavaScript é uma abordagem de desenvolvimento de software que visa tornar o código mais robusto e resistente a erros. O objetivo é antecipar possíveis problemas e tomar medidas preventivas para evitar falhas ou comportamentos indesejados.

Na programação defensiva, são adotadas práticas e técnicas para lidar com situações inesperadas ou entradas inválidas de forma segura. Isso inclui a validação de dados de entrada, o tratamento adequado de erros e a implementação de mecanismos de segurança.

Algumas práticas comuns de programação defensiva em JavaScript incluem:

- Validação de entrada: Verificar se os dados de entrada estão no formato esperado e atendem aos requisitos necessários antes de processá-los. Isso pode envolver a verificação de tipos de dados, a validação de valores mínimos ou máximos e a prevenção de injeção de código malicioso.
- Tratamento de erros: Utilizar blocos "try-catch" para capturar e tratar erros de forma adequada. Isso permite que você identifique e lide com erros de maneira controlada, evitando que o programa pare de funcionar ou exiba mensagens de erro confusas para o usuário.
- Verificação de existência: Antes de acessar propriedades de objetos ou chamar métodos, verificar se eles existem para evitar erros de referência. Isso pode ser feito usando condicionais ou operadores de verificação de nulos ou indefinidos.
- Tratamento de exceções: Identificar e tratar exceções específicas que podem ocorrer durante a execução do código. Isso permite que você tome ações apropriadas para lidar com situações excepcionais, como erros de rede, falhas de banco de dados ou erros de API.
- Uso de padrões de projeto: Utilizar padrões de projeto que promovam a segurança e a robustez do código, como o padrão Singleton, o padrão Observer ou o padrão Strategy. Esses padrões podem ajudar a separar preocupações, facilitar a manutenção e melhorar a resiliência do código.

ERROS

Erros em JavaScript são problemas que ocorrem durante a execução de um programa escrito em JavaScript. Eles podem ser causados por erros de sintaxe, erros de lógica ou condições inesperadas durante a execução do código.

Existem diferentes tipos de erros em JavaScript, incluindo:

- **Erros de sintaxe:** Esses erros ocorrem quando o código JavaScript viola as regras de sintaxe da linguagem. Por exemplo, esquecer de fechar um parêntese ou usar uma palavra-chave reservada de forma incorreta.
- **Erros de referência:** Esses erros ocorrem quando uma variável ou função é referenciada, mas não foi definida anteriormente. Isso pode acontecer quando uma variável é digitada incorretamente ou quando uma função não está acessível no escopo atual.
- **Erros de tipo:** Esses erros ocorrem quando uma operação é realizada em um tipo de dado incompatível. Por exemplo, tentar somar uma string com um número ou chamar um método em um objeto que não possui esse método.
- **Erros de lógica:** Esses erros ocorrem quando o código JavaScript não produz o resultado esperado devido a uma falha na lógica do programa. Isso pode acontecer quando uma condição não é avaliada corretamente ou quando um loop não é encerrado adequadamente.
- **Erros de tempo de execução:** Esses erros ocorrem durante a execução do código JavaScript devido a condições inesperadas, como divisão por zero, acesso a um elemento de matriz inexistente ou chamada de função com argumentos inválidos.

Quando ocorre um erro em JavaScript, geralmente é exibida uma mensagem de erro no console do navegador, que pode ajudar a identificar a causa do problema. É importante entender os diferentes tipos de erros em JavaScript para poder depurar e corrigir problemas em seu código.

TRATAMENTO DE ERROS

O tratamento de erros em JavaScript é uma técnica utilizada para lidar com erros que podem ocorrer durante a execução de um programa. Ele permite que você controle o fluxo do programa e tome ações apropriadas quando um erro é encontrado, em vez de permitir que o erro interrompa a execução do programa abruptamente.

O tratamento de erros em JavaScript é uma prática recomendada para garantir que seu programa seja mais robusto e resiliente a erros. Ele permite que você identifique e lide com problemas de forma adequada, melhorando a experiência do usuário e facilitando a depuração e manutenção do código.

Vamos abordar o tratamento de erros com o programa que criamos algumas aulas passadas, o Pythagora's Web Page, ele era responsável por calcular a Hipotenusa de um triângulo retângulo, e o código que construímos e usaremos como referência serão os códigos abaixo (index.html e script.js).

Calcule a Hipotenusa de qualquer Triângulo Retângulo

Primeiro Cateto

Segundo Cateto

Calcular Hipotenusa

```

index.html > ...
1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="UTF-8" />
5      <meta http-equiv="X-UA-Compatible" content="IE=edge" />
6      <meta name="viewport" content="width=device-width, ini
7      <title>Pythagoras' Web Page</title>
8      <link rel="stylesheet" href="style.css" />
9    </head>
10   <body>
11     <h2>Calcule a Hipotenusa de qualquer Triângulo Retângulo</h2>
12     <div class="answer-container">
13       
18       <div id="answer" class="hidden-element"></div>
19     </div>
20     <div class="inputs-container">
21       <label for="side-1">Primeiro Cateto</label>
22       <input id="side-1" type="text" />
23       <label for="side-2">Segundo Cateto</label>
24       <input id="side-2" type="text" />
25       <button id="calculate-hypotenuse">Calcular Hipotenusa</button>
26     </div>
27     <script src="/calculatePythagoras.js"></script>

```

```

calculatePythagoras.js > onCalculateButtonClick
1  function calculateSquareRoot(radicand) {
2    return radicand ** (1 / 2);
3  }
4
5  function calculatePythagoras(side1, side2) {
6    const sum = side1 ** 2 + side2 ** 2;
7    return calculateSquareRoot(sum);
8  }
9
10 const firstSideRef = document.getElementById('side-1');
11 const secondSideRef = document.getElementById('side-2');
12 const calculateButton = document.getElementById('calculate-hypotenuse');
13 const answerDiv = document.getElementById('answer');
14
15 function onCalculateButtonClick() {
16   const firstSide = firstSideRef.value;
17   const secondSide = secondSideRef.value;
18   // console.log('function executed');
19   answerDiv.classList.remove('hidden-element');
20   answerDiv.innerHTML = `A hipotenusa de um triângulo de lados ${firstSide}, ${secondSide} é: ${(firstSide ** 2 + secondSide ** 2).toFixed(3)}`.replace(/[^.]\000$/,'');
21   answerDiv.innerHTML = `A hipotenusa de um triângulo de lados ${firstSide}, ${secondSide} é: ${(firstSide ** 2 + secondSide ** 2).toFixed(3)}`.replace(/[^.]\000$/,'');
22 }
23
24 .toFixed(3)
25 .replace(/[^.]\000$/,'')
26 <div class="left-point"></div>;
27 }
28

```



Antecipar erros com condicionais em JavaScript é uma prática comum para evitar que erros ocorram durante a execução do código. Isso envolve a verificação de certas condições antes de executar determinadas ações, a fim de prevenir erros ou comportamentos indesejados.

Ao utilizar condicionais, você pode verificar se certas condições são atendidas antes de executar um bloco de código. Isso permite que você tome decisões com base nas condições e evite erros que possam ocorrer devido a valores inválidos ou inesperados.

- Verificar se os valores inseridos é uma string vazia (""): Se os valores dos catetos forem vazios, não podemos realizar o cálculo da hipotenusa.
- Verificar se os valores inseridos são números válidos: Podemos utilizar a função isNaN() para verificar se os valores inseridos são números. Se algum dos valores não for um número válido, podemos exibir uma mensagem de erro para o usuário.

```
if(firstSideRef.value === "" || secondSideRef.value === "" || isNaN(firstSideRef.value) || isNaN(secondSideRef.value)){  
}
```

isNaN()

A **função isNaN()** em JavaScript é usada para verificar se um valor não é um número (NaN). Ela retorna true se o valor passado como argumento não for um número e false caso contrário.

O termo "**NaN**" significa "Not a Number" (Não é um número) e é um valor especial em JavaScript que representa um resultado inválido ou indefinido em operações matemáticas.

A **função isNaN()** é útil quando você precisa verificar se um valor é um número válido antes de realizar operações matemáticas ou quando precisa validar a entrada do usuário para garantir que seja um número.

Ao utilizar a **função isNaN()**, é importante lembrar que ela pode retornar true para valores que não são números, mas também para outros casos, como strings vazias ou valores booleanos. Portanto, é necessário considerar o contexto em que a função está sendo utilizada e realizar as verificações adequadas para garantir a validade dos dados.

Console.error()

A **função console.error()** em JavaScript é usada para exibir mensagens de erro no console do navegador ou no console de desenvolvimento. Ela é especialmente útil para registrar erros e problemas durante a execução do código.

Quando você chama **console.error()**, uma mensagem de erro é exibida no console com um ícone de erro ao lado dela. Essa mensagem pode conter informações sobre o erro, como uma descrição do problema, uma mensagem de erro específica ou até mesmo uma pilha de chamadas de função que levaram ao erro.

A **função console.error()** é útil para depurar e identificar problemas em seu código. Ao exibir mensagens de erro, você pode obter informações valiosas sobre o que deu errado e onde ocorreu o erro. Isso pode ajudar a localizar e corrigir problemas mais rapidamente durante o processo de desenvolvimento.

É importante mencionar que a função **console.error()** não interrompe a execução do código. Ela apenas registra o erro no console.

Return

Em seguida, a instrução `return` é executada sem nenhum valor especificado. Isso significa que a função em que esse código está sendo executado será encerrada imediatamente, sem retornar nenhum valor.

O uso do `return` sem um valor é comum quando você deseja encerrar a execução de uma função sem retornar um resultado específico. É útil em situações em que você precisa sair prematuramente de uma função ou quando não há necessidade de retornar um valor específico.

O `return` é usado logo após a exibição da mensagem de erro. Isso indica que, após exibir a mensagem de erro, a função será encerrada imediatamente, sem executar qualquer código adicional que possa estar presente na função.

```
if (
  firstSideRef.value === '' ||
  secondSideRef.value === '' ||
  isNaN(firstSideRef.value) ||
  isNaN(secondSideRef.value)
) {
  console.error(
    'Entradas inválidas. Os tamanhos dos catetos devem ser fornecidos, e seus valores devem ser números positivos exclusivamente!'
  );
  return;
}
```

TRY / CATCH

O bloco try...catch em JavaScript é uma estrutura que permite capturar e tratar erros durante a execução do código. Ele é usado para envolver um trecho de código que pode gerar exceções, permitindo que você lide com essas exceções de forma controlada. O bloco try...catch consiste em dois componentes principais: o bloco try e o bloco catch.

O bloco try é onde você coloca o código que pode gerar uma exceção. Dentro desse bloco, você pode incluir qualquer código que possa lançar uma exceção, como operações matemáticas, chamadas de função ou acesso a propriedades de objetos.

Se ocorrer uma exceção dentro do bloco try, o fluxo de execução é interrompido e o controle é transferido para o bloco catch. O bloco catch é onde você pode tratar a exceção capturada e executar um código específico para lidar com o erro.

A estrutura básica do bloco try...catch é a seguinte:

```
try {  
    // Código que pode gerar uma exceção  
} catch (erro) {  
    // Código para tratar a exceção  
}
```



Quando uma exceção é lançada dentro do bloco try, o controle é transferido para o bloco catch e o erro é capturado na variável erro. Você pode nomear essa variável como desejar, mas é comum usar o nome erro ou exceção para deixar claro o seu propósito.

Dentro do bloco catch, você pode executar qualquer código necessário para lidar com a exceção. Isso pode incluir exibir mensagens de erro, registrar informações sobre o erro, tomar ações alternativas ou até mesmo lançar uma nova exceção.

O bloco try...catch é útil para lidar com erros de forma controlada, permitindo que você tome ações apropriadas quando exceções ocorrem. Ele ajuda a evitar que o código pare de funcionar completamente quando um erro ocorre e permite que você forneça uma experiência mais robusta e amigável para o usuário.

É importante mencionar que o bloco try...catch deve ser usado com cuidado e apenas para capturar exceções específicas que você espera e sabe como lidar. Capturar todas as exceções indiscriminadamente pode dificultar a depuração de erros e ocultar problemas reais no código.

```
function onCalculateButtonClick() {
  try {
    const firstSide = firstSideRef.value;
    const secondSide = secondSideRef.value;
    answerDiv.classList.remove('hidden-element');
    answerDiv.innerHTML = `A hipotenusa de um triangulo de lados ${firstSide} e ${secondSide} é: ${calculatePythagoras(
      firstSide,
      secondSide
    )}
    .toFixed(3)
    .replace(/[,]000$/, '')
  <div class="left-point"></div>`;
  } catch (error) {
    console.log(error);
  }
}
```



THROW

A palavra-chave `throw` em JavaScript é usada para lançar (ou "throw", em inglês) uma exceção manualmente. Ela permite que você crie e lance suas próprias exceções personalizadas durante a execução do código.

Quando você usa a palavra-chave `throw`, você está sinalizando que ocorreu uma condição excepcional ou um erro específico no seu código. Isso pode ser útil quando você precisa lidar com situações excepcionais que não podem ser tratadas de forma adequada pelo fluxo normal do programa.

O uso do `throw` permite que você crie exceções personalizadas e forneça informações detalhadas sobre o erro. Isso pode ser útil para comunicar problemas específicos no código, facilitando a depuração e o tratamento adequado de erros.

É importante mencionar que, ao lançar uma exceção com `throw`, o fluxo de execução do código é interrompido imediatamente. Portanto, é necessário garantir que você esteja capturando e tratando a exceção adequadamente usando um bloco `try...catch` ou permitindo que a exceção seja propagada para um bloco superior.

```
function onCalculateButtonClick() {
  try {
    const firstSide = firstSideRef.value;
    const secondSide = secondSideRef.value;
    if (
      firstSide === '' ||
      secondSide === '' ||
      isNaN(firstSide) ||
      isNaN(secondSide) ||
      firstSide > 0 ||
      secondSide > 0
    ) {
      throw 'Entradas inválidas. Os tamanhos dos catetos devem ser fornecidos, e seus valores devem ser números positivos exclusivamente!';
    }
    answerDiv.classList.remove('hidden-element');
    answerDiv.innerHTML = `A hipotenusa de um triangulo de lados ${firstSide} e ${secondSide} é: ${calculatePythagoras(
      firstSide,
      secondSide
    )}
      .toFixed(3)
      .replace(/[,]000$/ , '')}`;
    <div class="left-point"></div>`;
  } catch (error) {
    console.log(error);
  }
}
```



Vamos realizar alguns ajustes no código, para que tratemos alguns cenários como valores numéricos decimais, conversões de String para Number e deixar nossas funções mais fortes separadamente, ou seja, cada função lidar com erros que podem acontecer dentro delas, da seguinte forma:

```
function onCalculateButtonClick() {
  try {
    const firstSide = Number(firstSideRef.value);
    const secondSide = Number(secondSideRef.value);
    if [
      isNaN(firstSide) ||
      isNaN(secondSide) ||
      firstSide <= 0 ||
      secondSide <= 0
    ] {
      throw 'Entradas inválidas. Os tamanhos dos catetos devem ser fornecidos, e seus valores devem ser números positivos exclusivamente!';
    }
  }
}
```

```
function calculateSquareRoot(radical) {
  try {
    if (radical < 0) {
      throw 'O radicando deve ser real e positivo';
    }
    return radical ** (1 / 2);
  } catch (error) [
    console.log(error);
  ]
}
```

```
function calculatePythagoras(side1, side2) {
  try {
    const sum = side1 ** 2 + side2 ** 2;
    return calculateSquareRoot(sum);
  } catch (error) {
    console.log(error);
  }
}
```

Ao lidar com tratamento de erros em JavaScript, é uma boa prática lançar um erro completo, incluindo uma mensagem de erro descritiva e, se possível, informações adicionais relevantes. Isso ajuda na depuração e no entendimento do erro, tanto para os desenvolvedores quanto para os usuários do seu código.

Ao lançar um erro completo, você fornece informações úteis sobre o que deu errado e por quê. Isso pode incluir detalhes sobre o contexto em que o erro ocorreu, valores de variáveis relevantes, pilha de chamadas de função e qualquer outra informação que possa ajudar a identificar e resolver o problema.

Lançar um erro completo com informações detalhadas é especialmente útil durante o desenvolvimento e a depuração do código, pois ajuda a identificar a causa raiz do problema. No entanto, ao implantar o código em um ambiente de produção, é importante ter cuidado para não expor informações sensíveis ou confidenciais nos erros lançados.

Throw New Error()

A expressão `throw new Error()` em JavaScript é usada para lançar (ou "throw", em inglês) um novo objeto de erro. Essa expressão cria uma instância da classe `Error` e a lança como uma exceção.

Quando você usa `throw new Error()`, você está criando um novo objeto de erro que pode conter uma mensagem descritiva sobre o erro que ocorreu. Essa mensagem é passada como argumento para o construtor da classe `Error`.

Ao lançar um novo objeto de erro, você pode personalizar a mensagem de erro para fornecer informações específicas sobre o problema que ocorreu. Isso ajuda na depuração e no entendimento do erro, tanto para os desenvolvedores quanto para os usuários do seu código.

Além disso, a classe Error em JavaScript possui outras propriedades e métodos que podem ser úteis ao lidar com erros. Por exemplo, a propriedade name contém o nome do tipo de erro (geralmente "Error"), e a propriedade stack contém uma representação da pilha de chamadas de função no momento em que o erro foi lançado.

```
function calculateSquareRoot(radicand) {
  if (radicand < 0) {
    throw new Error('O radicando deve ser real e positivo');
  }
  return radicand ** (1 / 2);
}
```

Além do tipo e mensagem do erro, ele também nos retorna o local que está ocorrendo o lançamento do erro:

```
Error: Entradas inválidas. Os tamanhos dos catetos devem ser fornecidos, e seus valores devem ser calculatePythagoras.js:45
números positivos exclusivamente!
at HTMLButtonElement.onCalculateButtonClick (calculatePythagoras.js:32:13)
```

Error: O radicando deve ser real e positivo at calculateSquareRoot (calculatePythagoras.js:3:11) at calculatePythagoras (calculatePythagoras.js:11:12) at HTMLButtonElement.onCalculateButtonClick (calculatePythagoras.js:37:100)	calculatePythagoras.js:13
TypeError: Cannot read properties of undefined (reading 'toFixed') at HTMLButtonElement.onCalculateButtonClick (calculatePythagoras.js:41:7)	calculatePythagoras.js:45

FINALLY

O **bloco finally** em JavaScript é usado em conjunto com os blocos try e catch para definir um trecho de código que será executado independentemente de ocorrer uma exceção ou não. Ele fornece uma maneira de garantir que determinadas ações sejam realizadas, independentemente do resultado do bloco try ou do tratamento de exceções no bloco catch.

A estrutura básica do bloco try...catch...finally é a seguinte:

```
try {  
    // Código que pode gerar uma exceção  
} catch (erro) {  
    // Código para tratar a exceção  
} finally {  
    // Código que será executado sempre  
}
```

O **bloco finally** é opcional, mas quando presente, ele será executado independentemente de ocorrer uma exceção ou não. Isso significa que o código dentro do bloco finally sempre será executado, seja após o bloco try ou após o bloco catch.

O **bloco finally** é útil para realizar ações que devem ser executadas independentemente do resultado do bloco try ou do tratamento de exceções. Isso pode incluir a liberação de recursos, como fechar conexões de banco de dados, encerrar arquivos abertos ou limpar variáveis temporárias.



O bloco `finally` é especialmente útil quando você precisa garantir que determinadas ações sejam realizadas, mesmo em caso de exceção. Ele permite que você mantenha um código limpo e organizado, separando a lógica de tratamento de exceções da lógica de ações que devem ser executadas sempre.

```
② bloco finally executado!                                         calculatePythagoras.js:47
Error: Entradas inválidas. Os tamanhos dos catetos devem ser fornecidos, e seus valores devem ser números positivos exclusivamente! calculatePythagoras.js:45
at HTMLButtonElement.onCalculateButtonClick (calculatePythagoras.js:32:13)
bloco finally executado!                                         calculatePythagoras.js:47
>
```

Módulo 14

PRIMEIRO PROJETO

PRIMEIRO PROJETO

PRIMEIRO PROJETO



Um projeto front-end envolvendo HTML, CSS e JavaScript é responsável pela criação e implementação da interface visual de um site ou aplicativo web.

Em um projeto front-end, o desenvolvedor utiliza três tecnologias em conjunto para criar uma interface visual atraente, responsiva e funcional. O HTML é utilizado para estruturar o conteúdo, o CSS é utilizado para estilizar e dar estilo à página, e o JavaScript é utilizado para adicionar interatividade e funcionalidade.

É importante mencionar que o front-end não se limita apenas a essas três tecnologias. Existem muitas outras ferramentas, bibliotecas e frameworks que podem ser utilizados para facilitar o desenvolvimento front-end, como por exemplo o Bootstrap. Essas ferramentas fornecem recursos adicionais e facilitam a criação de projetos mais complexos e eficientes.

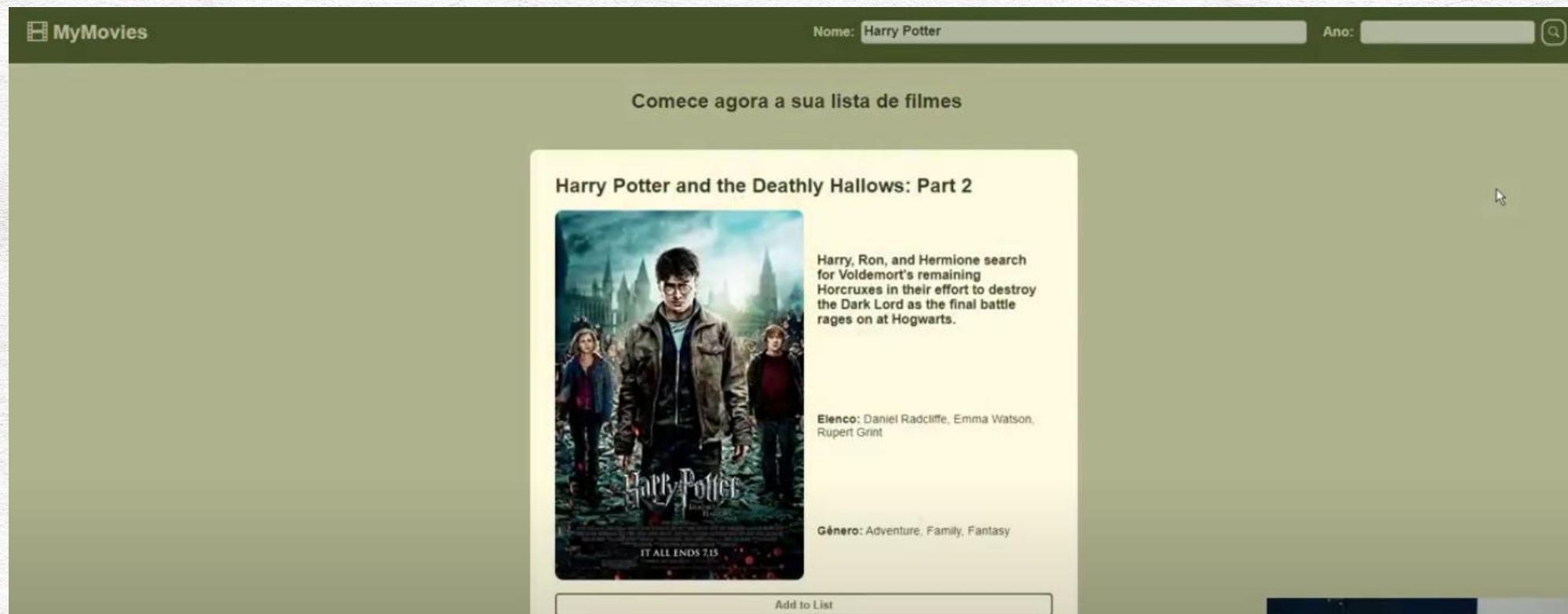
Em nosso projeto web, iremos utilizar um serviço de backend que está disponível na internet. Isso significa que iremos utilizar uma parte do nosso projeto que será responsável por processar e fornecer os dados necessários para o funcionamento da nossa aplicação. O backend é responsável por realizar operações complexas, como a manipulação de dados, autenticação de usuários e a comunicação com bancos de dados. Utilizando um serviço de backend existente, podemos aproveitar funcionalidades já implementadas e focar no desenvolvimento da interface visual e na interação com o usuário no front-end. Essa divisão de responsabilidades entre o front-end e o backend é comum em projetos web e permite uma melhor organização e escalabilidade do projeto.

O projeto consiste em criar um gerenciador de coleção de filmes, onde será possível catalogar e organizar os filmes de forma fácil e prática. O objetivo é permitir que os usuários tenham um controle eficiente sobre sua coleção.



Para melhorar a experiência do usuário, vamos implementar recursos de busca e filtro, permitindo que o usuário localize facilmente os filmes cadastrados em sua coleção, através do nome e ano do filme.

Podendo adicionar ou remover os filmes da sua coleção.



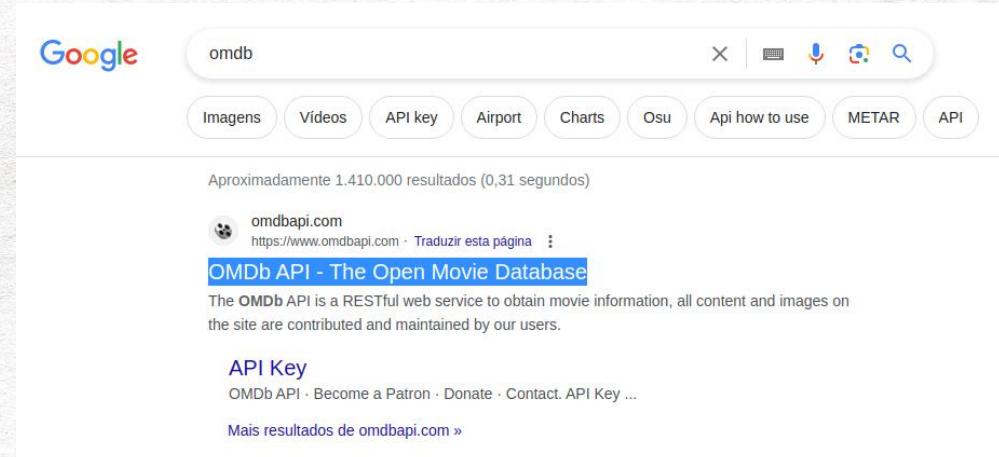
APIs

Uma **API** disponibilizada na internet é uma interface de programação que permite a comunicação entre diferentes softwares. Ela define como os softwares devem interagir e trocar informações entre si. As APIs são utilizadas para acessar serviços, integrar sistemas e obter dados de forma segura e eficiente. Ao utilizar uma API disponibilizada na internet, é necessário seguir a documentação fornecida pelo provedor da API para utilizar corretamente seus recursos.

APIs REST são mais flexíveis e populares na web atualmente. Elas permitem que o cliente envie solicitações ao servidor na forma de dados e o servidor utiliza esses dados para executar suas funções internas e retornar os dados de saída ao cliente.

Iremos utilizar a **OMDb API - The Open Movie Database**, que é uma API (Interface de Programação de Aplicativos) que fornece informações sobre filmes, séries de televisão e outros conteúdos relacionados. Ela permite que os desenvolvedores acessem dados como título, ano de lançamento, elenco, diretor, sinopse, classificação, avaliações e muito mais.

Primeiro, faça uma busca pelo nome **OMDb**, será o primeiro link que o google retornará:



O site oficial da API mostra como que podemos utilizar ela:

The screenshot shows the OMDB API Usage page. At the top, there is a navigation bar with links: OMDB API, Usage, Parameters, Examples, Change Log, API Key, Become a Patron, Donate, and Contact. Below the navigation bar, there is a list of websites and services that use the OMDB API, including Emby, Trakt, FileBot, Reelgood, Xirvik Servers, Yidio, mi.tv, Couchpop, What's on Netflix, Edu Reviewer, Flixboss, StreamingMoviesRight, Scripts on Screen, Writers Per Hour, Medium.com, Write my paper, Ramotion.com, Phone Trackers, Property for sale in Lake Como, iStarTips, What A Room, Vibelovely, StreamToday, and Property for sale in Spain. The main content area is titled "Usage" and contains two examples of API URLs. The first example, "Send all data requests to:", shows the URL `http://www.omdbapi.com/?apikey=[yourkey]&`. The second example, "Poster API requests:", shows the URL `http://img.omdbapi.com/?apikey=[yourkey]&`.

Para utilizar a OMDB API, é necessário fazer uma solicitação HTTP para o endpoint correto da API, passando os parâmetros necessários, como o título do filme ou série desejada. A resposta da API será retornada em formato JSON por padrão, mas também é possível solicitar uma resposta em formato XML. A resposta da API conterá os dados solicitados sobre o filme ou série, que podem ser utilizados pelo desenvolvedor em seu projeto.

Quando dizemos endpoint correto da API e passar parâmetros, estamos falando sobre query string parameter:

Um **query string parameter**, em português, é um parâmetro presente em uma URL que é utilizado para enviar informações adicionais para um servidor web. Ele é composto por um nome e um valor, separados por um sinal de igual (=), e é adicionado à URL após o ponto de interrogação (?).

A função do query string parameter é fornecer dados extras para o servidor, que pode utilizar essas informações para processar a requisição de forma apropriada.

Na OMDB API precisamos fazer o cadastro no site, para gerarmos a **chave da API (apikey)**:

Send all data requests to:

`http://www.omdbapi.com/?apikey=[yourkey]&`

Poster API requests:

`http://img.omdbapi.com/?apikey=[yourkey]&`



Acesse a aba API KEY

OMDb API Usage Parameters Examples Change Log API Key

API Key

10/05/22 Email Delay! If your requested key doesn't show up within an hour, please contact me directly.

Generate API Key

Account Type Patreon FREE! (1,000 daily limit)

Email

Name First Name Last Name

Use
A short description of the application or website that will use this API.

Submit

Faça o cadastro de forma gratuita, dessa forma você terá autorização de realizar 1000 requisições por dia, ou seja, uma requisição para uma API é uma solicitação feita por um cliente a um servidor para obter ou enviar dados, a API receberá até 1000 requisições ao dia, e processará as informações e retornará as respostas contendo os dados solicitados ou uma confirmação de cada operação realizada.

Você receberá a chave no email cadastrado, copie o mesmo pois iremos utilizá-lo em outro momento no nosso código.

Não devemos compartilhar uma chave de API com outras pessoas porque isso coloca em risco a segurança da nossa conta e dos dados acessados pela API. As chaves de API são semelhantes a senhas e devem ser tratadas com cuidado. Compartilhar uma chave de API pode resultar em acesso não autorizado, perdas financeiras e violações de segurança. É importante proteger nossas chaves de API e, se comprometidas, desativá-las imediatamente.

BIBLIOTECAS

Uma biblioteca em JavaScript é um conjunto de código pré-escrito e disponibilizado para uso em projetos de desenvolvimento web. Essas bibliotecas contêm funções, classes e métodos que podem ser utilizados para realizar tarefas específicas de forma mais fácil e eficiente. Elas são criadas para resolver problemas comuns e fornecer funcionalidades adicionais que podem auxiliar no desenvolvimento de um projeto.

As bibliotecas em JavaScript são desenvolvidas para serem reutilizáveis, ou seja, podem ser utilizadas em diferentes projetos, economizando tempo e esforço na implementação de funcionalidades comuns. Elas são criadas por desenvolvedores e disponibilizadas para a comunidade, permitindo que outros programadores as utilizem em seus projetos.

BIBLIOTECA DE ÍCONES

Essas bibliotecas contêm uma variedade de ícones que representam diferentes conceitos, objetos ou ações, como setas, símbolos de redes sociais, ícones de menu, entre outros.

Essas bibliotecas de ícones são muito úteis para desenvolvedores, pois permitem que eles utilizem ícones prontos em seus projetos, economizando tempo e esforço no design e criação de ícones personalizados. Além disso, as bibliotecas de ícones costumam fornecer uma ampla variedade de estilos, tamanhos e formatos de ícones, permitindo que os desenvolvedores escolham os que melhor se adequam ao visual e às necessidades de seus projetos.

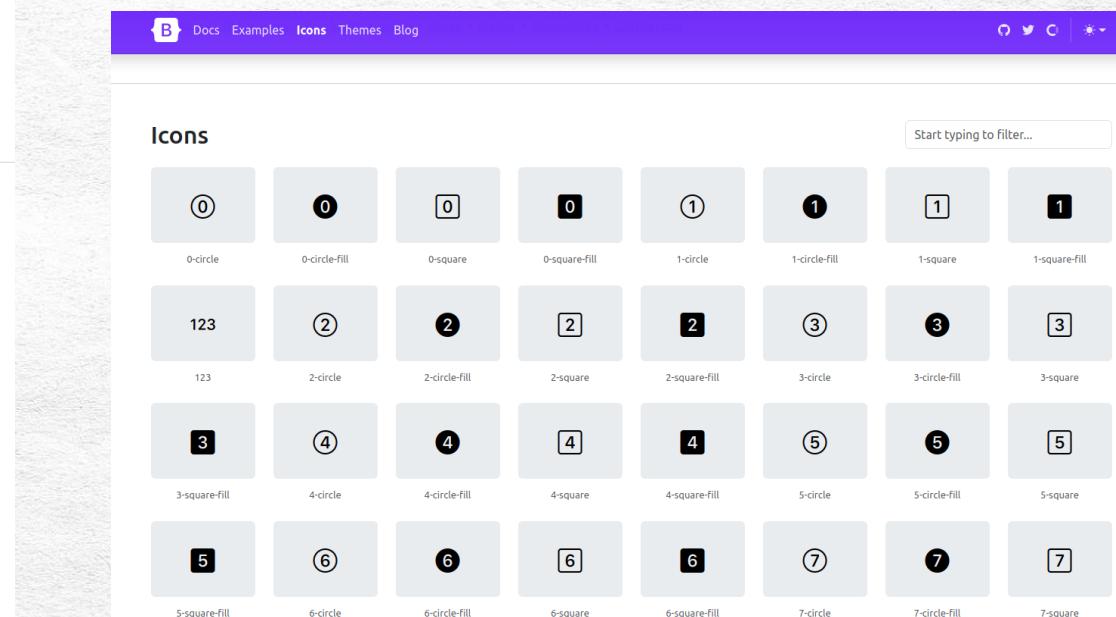
Bootstrap Icons

Bootstrap Icons é uma biblioteca de ícones, fornece uma coleção de ícones vetorizados em formato SVG.

Os ícones do Bootstrap Icons são projetados para serem escaláveis e acessíveis. Eles são fornecidos no formato SVG, o que significa que podem ser redimensionados sem perder qualidade ou nitidez. Além disso, o formato SVG oferece suporte a recursos de acessibilidade, tornando os ícones mais amigáveis para pessoas com deficiência visual.

Pesquise no google, e entre no site oficial:

A screenshot of a Google search results page for "bootstrap icons". The search bar at the top contains the query. Below it, there are several filter buttons: "Imagens", "CDN", "React", "Download", "How to use", "Npm", "Angular", "Link", and "Size". The main search results area shows a snippet from the official Bootstrap Icons website, which includes a link to "getbootstrap.com" and a brief description of the library. Below this, there are sections for "Icons", "Search", "List", and "Bootstrap Icons v1.10.0". Each section has a brief description and a link to the official documentation.

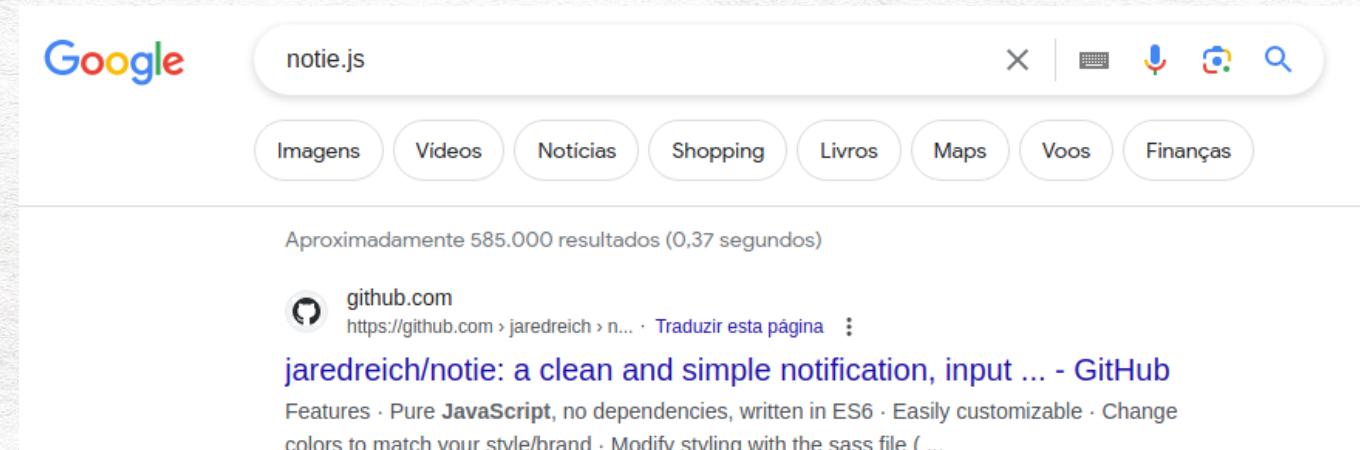


Notie.js

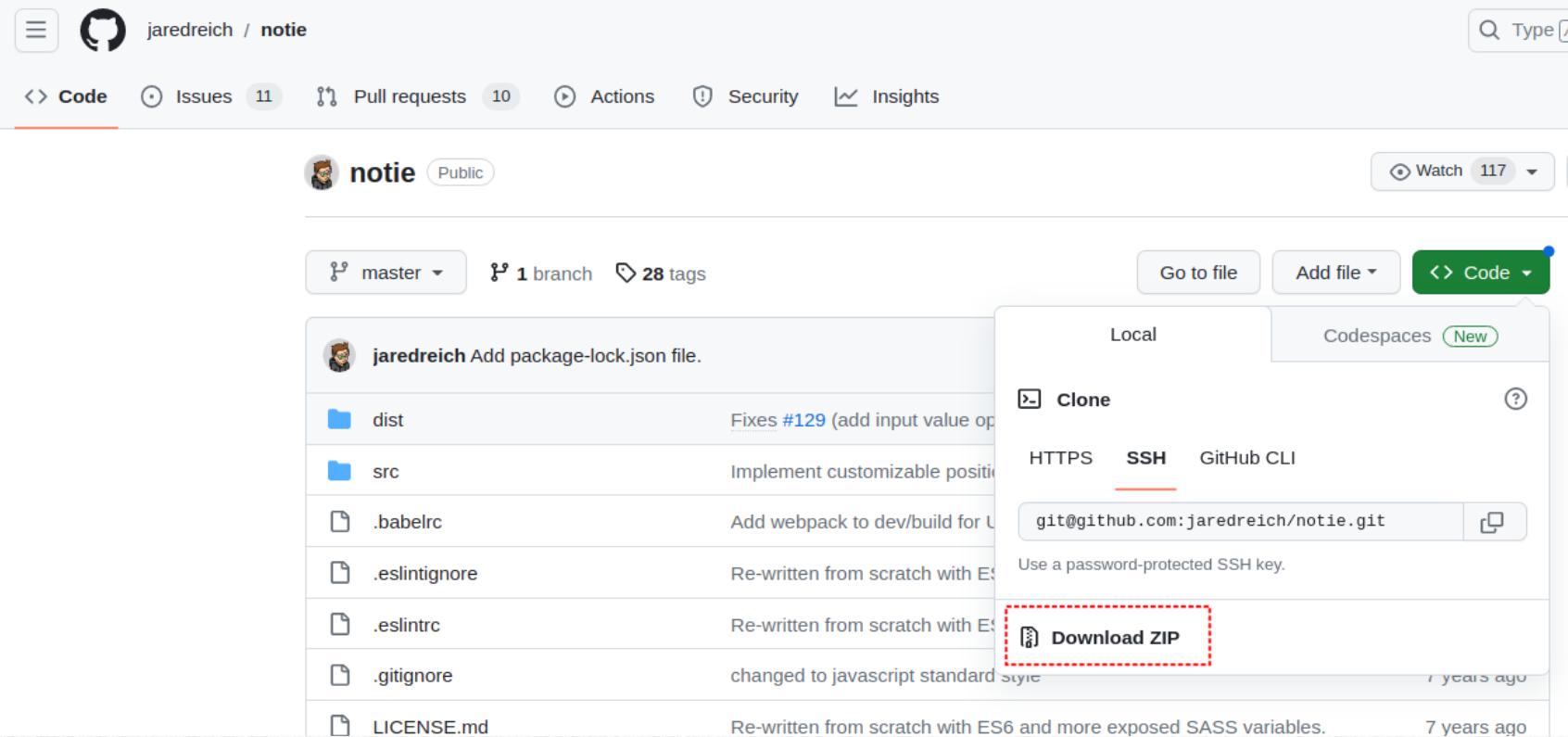
O Notie.js é uma biblioteca JavaScript leve e simples que permite exibir notificações personalizadas em um site ou aplicativo web. Com o Notie.js, é possível exibir mensagens de sucesso, erro, aviso ou informações para o usuário de forma elegante e não intrusiva.

Essa biblioteca é útil quando se deseja fornecer feedback visual ao usuário, como uma confirmação de envio de um formulário, um aviso sobre uma ação realizada ou uma mensagem de erro em caso de problemas. As notificações do Notie.js podem ser exibidas em diferentes posições na tela, como no canto superior direito ou inferior direito, e possuem animações suaves para uma experiência de usuário agradável.

Pesquise no google, e vamos acessar o gitHub do desenvolvedor dessa biblioteca:



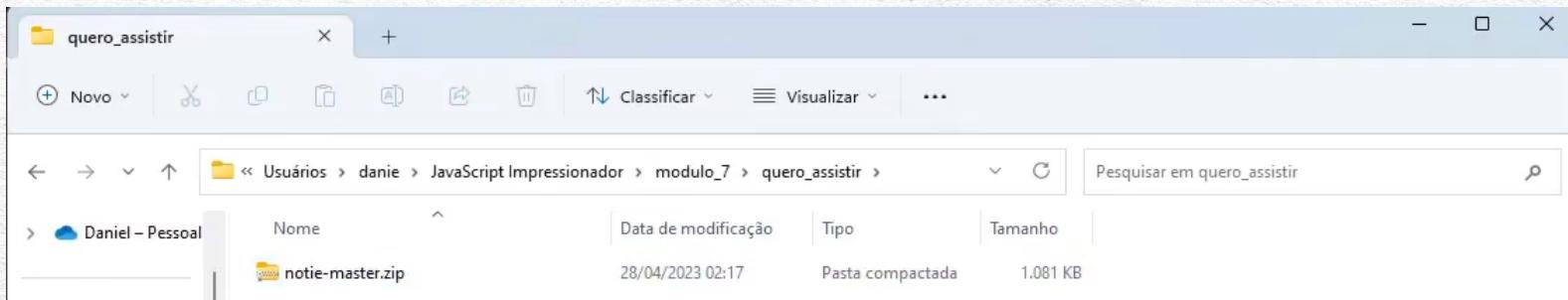
Clique em Code e faça o download do arquivo ZIP:



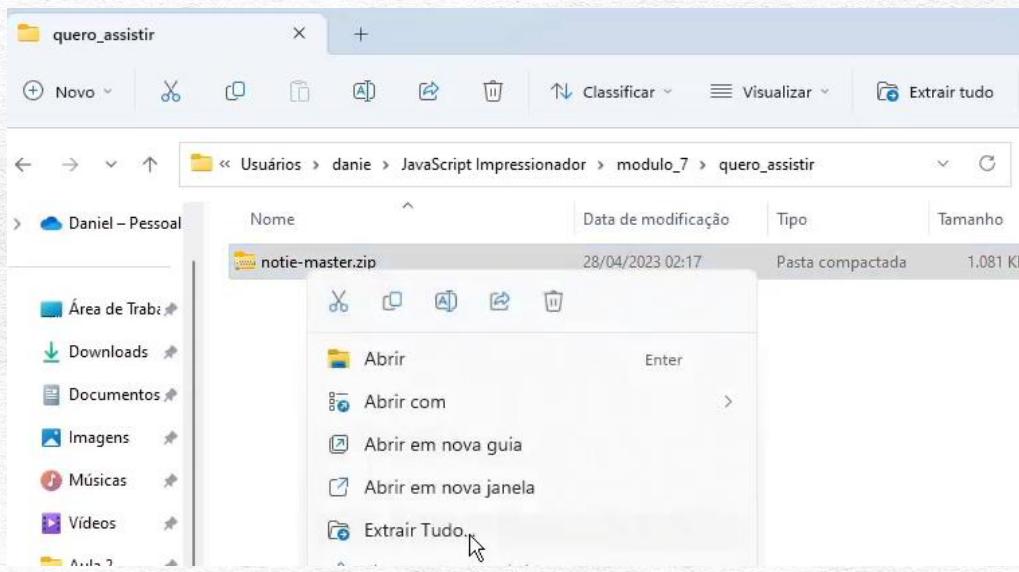
Depois iremos adicionar a pasta ao nosso projeto.

Módulo 14 – Começando a estrutura do projeto (1 / 7)

Para iniciarmos o projeto crie uma pasta chamada **quero_assistir** e dentro dela cole a **pasta da biblioteca NOTIE** que baixamos na aula anterior.



Clique com o botão direito no arquivo e extraia o conteúdo do mesmo.



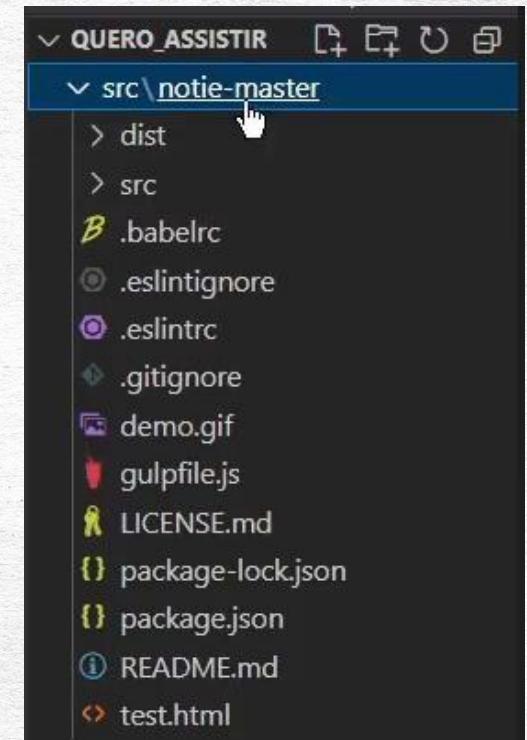
Pasta SRC

A pasta "src" em projetos JavaScript é uma convenção comum usada para armazenar o código-fonte do projeto. "src" é uma abreviação de "source" (fonte em inglês) e é usada para distinguir a pasta que contém o código-fonte dos arquivos de configuração, testes e outros recursos do projeto.

Dentro da pasta "src", você geralmente encontrará os arquivos JavaScript que contêm a lógica principal do seu projeto. Esses arquivos podem incluir funções, classes, variáveis e outros elementos do código que são necessários para que o projeto funcione corretamente.

A estrutura exata da pasta "src" pode variar dependendo do projeto e das preferências do desenvolvedor. Alguns projetos podem ter subpastas dentro de "src" para organizar o código em módulos ou componentes. Outros projetos podem ter todos os arquivos JavaScript diretamente dentro da pasta "src".

Nesse momento iremos colocar a pasta extraída da biblioteca notie, dentro da pasta src.



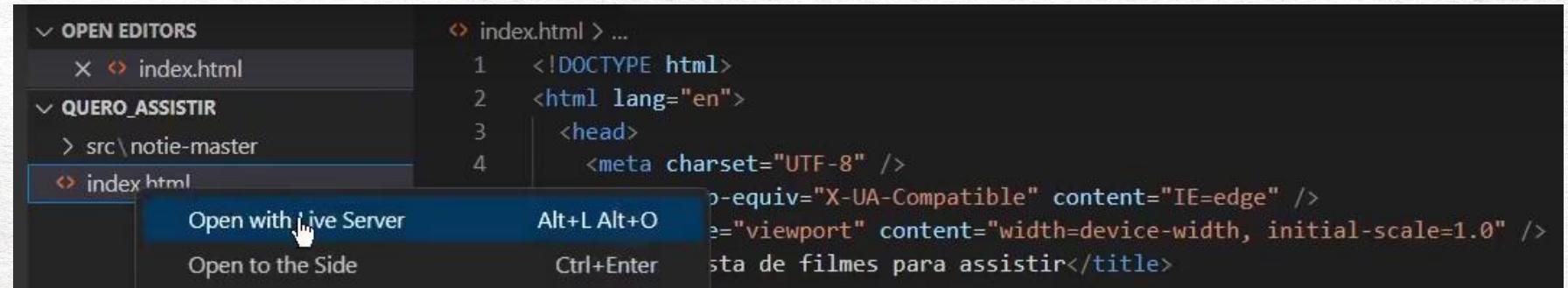
Iniciaremos a estrutura do HTML:



The screenshot shows the VS Code interface. The Explorer sidebar on the left lists a project folder 'QUERO_ASSISTIR' containing a 'src' folder with 'notie-master' and an 'index.html' file. The Editor pane on the right displays the code for 'index.html':

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<meta http-equiv="X-UA-Compatible" content="IE=edge" />
<meta name="viewport" content="width=device-width, initial-scale=1.0" />
<title>Lista de filmes para assistir</title>
</head>
<body></body>
</html>
```

E abriremos o nosso navegador através do LIVE SERVER:



Elemento header

A **tag <header>** em HTML é usada para definir o cabeçalho de uma seção ou de todo o documento HTML. Ela é usada para agrupar elementos que representam informações introdutórias ou de navegação no topo de uma página.

O conteúdo dentro da **tag <header>** geralmente inclui elementos como logotipos, títulos, menus de navegação e outros elementos que fornecem informações importantes sobre o conteúdo da página ou do site.

A **tag <header>** é uma das tags semânticas introduzidas no HTML5, o que significa que ela tem um significado específico e ajuda a estruturar o conteúdo de forma mais clara e significativa para os mecanismos de busca e para os leitores de tela.

É importante notar que a **tag <header>** pode ser usada várias vezes em um documento HTML, dependendo da estrutura e do layout da página. Por exemplo, um site pode ter um cabeçalho principal que contém o logotipo e o menu de navegação, e também pode ter cabeçalhos menores em seções individuais da página.

Elemento main

A **tag <main>** em HTML é usada para definir o conteúdo principal de uma página. Ela representa a seção central e mais importante do conteúdo de um documento HTML.

A **tag <main>** é uma das tags semânticas introduzidas no HTML5, o que significa que ela tem um significado específico e ajuda a estruturar o conteúdo de forma mais clara e significativa para os mecanismos de busca e para os leitores de tela.

O conteúdo dentro da **tag <main>** geralmente inclui o conteúdo principal da página, como artigos, seções, blocos de texto, imagens e outros elementos que são essenciais para o propósito da página.

É importante notar que um documento HTML deve ter apenas uma tag <main>, que deve ser usada para envolver todo o conteúdo principal da página. A tag <main> não deve ser usada dentro de outras tags semânticas, como <article>, <section> ou <aside>, pois ela já representa o conteúdo principal da página.

A **tag <main>** também é útil para acessibilidade, pois ajuda os leitores de tela a identificar e navegar diretamente para o conteúdo principal de uma página.

```
<body>
  <header></header>
  <main></main>
</body>
```



Adicionaremos o texto MyMovies dentro da tag header e utilizaremos a biblioteca bootstrap icon para colocarmos os ícones de filme e de busca.

Para isso devemos entrar no site oficial da biblioteca e utilizar o modelo CDN, para utilizar os ícones no nosso projeto. Para isso você deve copiar o código do site e adicionar ao head do arquivo index.html.

CDN

Include the icon fonts stylesheet—in your website <head> or via `@import` in CSS—from jsDelivr and get started in seconds. [See icon font docs](#) for examples.

```
<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap-icons@1.10.5/font/bootstrap-icons.css" type="text/css"/>
```

```
@import url("https://cdn.jsdelivr.net/npm/bootstrap-icons@1.10.5/font/bootstrap-icons.css");
```

```
index.html > html > head > link
1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="UTF-8" />
5      <meta http-equiv="X-UA-Compatible" content="IE=edge" />
6      <meta name="viewport" content="width=device-width, initial-scale=1.0" />
7      <title>Lista de filmes para assistir</title>
8      <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap-icons@1.10.5/font/bootstrap-icons.css" type="text/css"/>
9    </head>
10   <body>
11     <header>MyMovies</header>
12     <main></main>
13   </body>
14 </html>
```

Pesquise pelo ícone movie / film, clique nele:

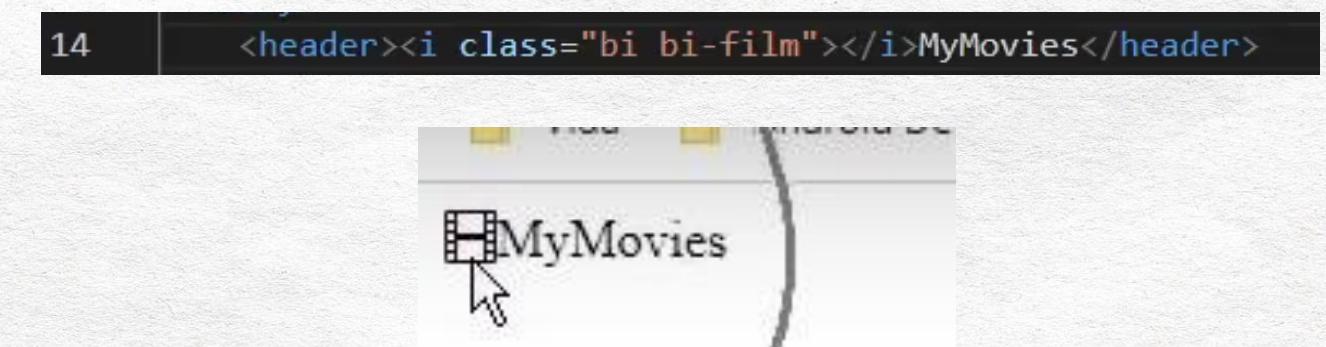


Copie o código de utilização do web font, e adicione dentro do elemento que você quer que esse ícone seja ligado. No caso, header.

Icon font

Using the web font? Copy, paste, and go.

```
<i class="bi bi-film"></i>
```



Na pasta base do nosso projeto iremos criar o nosso arquivo CSS para iniciarmos a estilização da nossa página.

```
↳ index.html > ⏷ html > ⏷ head > ⏷ link
1  <!DOCTYPE html>
2  <html lang="en">
3  |  <head>
4  |  |  <meta charset="UTF-8" />
5  |  |  <meta http-equiv="X-UA-Compatible" content="IE=edge" />
6  |  |  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
7  |  |  <title>Lista de filmes para assistir</title>
8  |  |  <link rel="stylesheet" href="./style.css">           I
9  |  |  <link
10 |  |  |  rel="stylesheet"
11 |  |  |  href="https://cdn.jsdelivr.net/npm/bootstrap-icons@1.10.5/font/bootstrap-icons.css"
12 |  |  |  />
13 |  |  </head>
14 |  |  <body>
15 |  |  |  <header><i class="bi bi-film"></i>MyMovies</header>
16 |  |  |  <main></main>
17 |  |  </body>
18 |  </html>
```

E vamos implementar um elemento h2 ao header que havíamos iniciado com o texto MyMovies:

```
<body>
  <header>
    |  <h2><i class="bi bi-film"></i>MyMovies</h2>
  </header>
  <main></main>
</body>
</html>
```



Elemento Button

A tag <button> em HTML é usada para criar um botão interativo em uma página da web. Ela permite que os usuários cliquem no botão para realizar uma ação ou executar um evento específico.

Vamos adicionar um botão ao lado do ícone de busca. Mais tarde, iremos adicionar a lógica de programação com JavaScript para que o botão possa realizar a busca de filmes. Por enquanto, vamos apenas estilizar o botão até alcançarmos o visual desejado do nosso projeto.

Para fazer isso, vamos seguir os mesmos passos que fizemos para adicionar o ícone de filme. Vamos acessar o site oficial do Bootstrap Icons, procurar pelo ícone de busca, clicar no ícone da lupa e copiar o código da fonte web para usar em nosso projeto.

The screenshot shows the Bootstrap Icons website interface. On the left, there's a grid of icons with labels: binoculars, binoculars-fill, google, search, search-heart, and search-heart-fill. The 'search' icon is highlighted with a cursor. Above the grid is a search bar with the placeholder 'search'. Below the grid, the heading 'Icon font' is displayed, followed by the text 'Using the web font? Copy, paste, and go.' A cursor is hovering over the copied HTML code: <i class="bi bi-search"></i>. To the right of the code editor, the HTML structure of the page is shown:

```
<body>
  <header>
    <h2><i class="bi bi-film"></i>MyMovies</h2>

    <button><i class="bi bi-search"></i></button>
  </header>
  <main></main>
</body>
```

Elemento Input

A **tag <input>** em HTML é usada para criar um campo de entrada interativo em uma página da web. Ela permite que os usuários insiram dados, como texto, números, datas, seleções de opções e muito mais.

A **tag <input>** possui vários tipos de atributos que determinam o comportamento e o tipo de entrada que será aceito. Alguns dos atributos mais comuns incluem:

- **type**: especifica o tipo de entrada que será aceito. Alguns valores possíveis são "**text**" (texto), "**number**" (número), "**date**" (data), "**checkbox**" (caixa de seleção), "**radio**" (botão de opção) e muitos outros.
- **name**: define um nome para o campo de entrada, que é usado para identificar o valor do campo quando o formulário é enviado.
- **value**: define um valor padrão para o campo de entrada.
- **placeholder**: exibe um texto de exemplo dentro do campo de entrada, fornecendo uma dica para os usuários sobre o que inserir.
- **required**: torna o campo de entrada obrigatório, exigindo que os usuários preencham o campo antes de enviar o formulário.

A tag <input> pode ser usada em diferentes contextos, como formulários, barras de pesquisa, campos de login e muito mais. Ela fornece uma maneira fácil de coletar dados dos usuários em uma página da web.

Elemento Label

A **tag <label>** em HTML é usada para associar um rótulo descritivo a um elemento de entrada, como um campo de texto ou uma caixa de seleção. Ela fornece uma descrição textual para o elemento de entrada, tornando-o mais acessível e compreensível para os usuários.

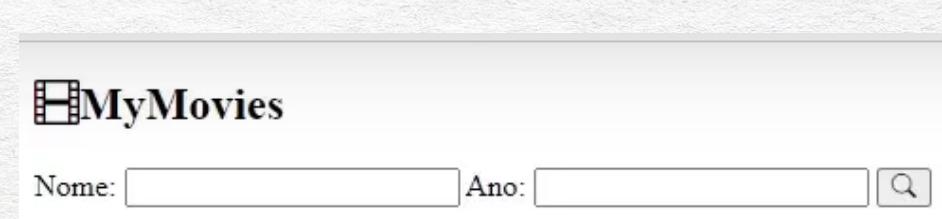
Ao usar a **tag <label>**, você pode envolver o texto descritivo entre as tags de abertura e fechamento.

O valor do atributo `for` deve corresponder ao valor do atributo `id` do elemento de entrada que está sendo rotulado. Isso permite que os usuários cliquem no rótulo para focar automaticamente no campo de entrada correspondente.

A **tag <label>** também pode ser usada para melhorar a acessibilidade, especialmente para usuários de leitores de tela. Ao associar um rótulo descritivo a um elemento de entrada, os leitores de tela podem fornecer uma descrição clara do campo para os usuários com deficiência visual.

É importante notar que o uso da **tag <label>** é uma prática recomendada em termos de usabilidade e acessibilidade. Ela ajuda a melhorar a experiência do usuário e torna o conteúdo da página mais compreensível.

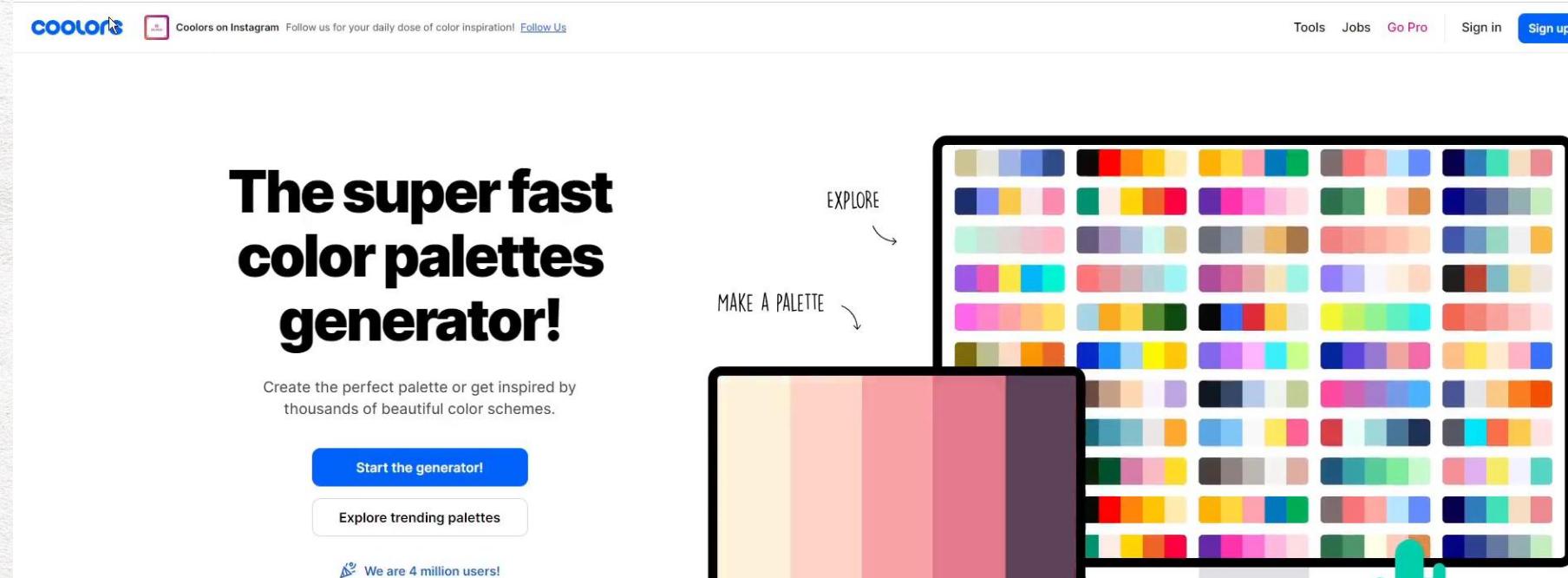
```
14  <body>
15    <header>
16      <h2><i class="bi bi-film"></i>MyMovies</h2>
17      <label for="movie-name">Nome:</label>
18      <input type="text" id="movie-name" />
19      <label for="movie-year">Ano:</label>
20      <input type="text" id="movie-year" />
21      <button><i class="bi bi-search"></i></button>
22    </header>
23    <main></main>
24  </body>
```



Vamos agrupar os elementos do tipo label e input dentro de uma div com a classe="input-wrapper".

```
14 <body>
15   <header>
16     <h2><i class="bi bi-film"></i>MyMovies</h2>
17     <div class="input-wrapper">
18       <label for="movie-name">Nome:</label>
19       <input type="text" id="movie-name" />
20     </div>
21     <div class="input-wrapper">
22       <label for="movie-year">Ano:</label>
23       <input type="text" id="movie-year" />
24     </div>
25     <button><i class="bi bi-search"></i></button>
26   </header>
27   <main></main>
28 </body>
```

Para escolher a paleta de cores que você quer estilizar o seu projeto, você pode acessar o site <https://coolors.co/> que ele te mostra várias opções e combinações.



:root em CSS

A declaração **:root {}** em CSS é usada para selecionar o elemento raiz de um documento HTML, ou seja, o elemento `<html>`. Ela permite definir variáveis CSS globais que podem ser usadas em todo o documento.

Dentro da declaração **:root {}**, você pode definir variáveis CSS usando a sintaxe **--nome-da-variavel: valor;**

O uso da declaração **:root {}** e das variáveis CSS globais permite criar um conjunto consistente de estilos que podem ser facilmente atualizados em todo o documento, tornando o código CSS mais modular e reutilizável.

```
# style.css > ...
1  :root {
2    --primary-color: #29335c;
3    --light-primary: #9ac4f8;
4    --lightest-color: #e9ebd5;
5    --secondary-color: #774936;
6    --light-secondary: #c08972;
7  }
8
```

No arquivo style.css iremos continuar a estilização do nosso Header:

```
9   * {  
10  |   box-sizing: border-box;  
11  |   margin: 0;  
12  |   padding: 0;  
13  |   font-family: sans-serif;  
14  }  
15
```

Seletor *

O seletor de CSS "*" é conhecido como *seletor universal*. Ele é usado para selecionar todos os elementos em um documento HTML. Quando você usa o seletor "*", ele corresponderá a todos os elementos HTML, independentemente do tipo ou da hierarquia.

Propriedade Box-sizing

A propriedade CSS box-sizing é usada para controlar como o tamanho total de um elemento é calculado, levando em consideração as bordas e o preenchimento (padding).

Existem dois valores principais para a propriedade box-sizing:

- **content-box** (valor padrão): Nesse modo, o tamanho total do elemento é calculado levando em consideração apenas o conteúdo do elemento. As bordas e o preenchimento são adicionados ao tamanho total do elemento. Isso significa que, se você definir a largura de um elemento como 200 pixels, por exemplo, as bordas e o preenchimento serão adicionados a essa largura, resultando em um tamanho total maior.
- **border-box**: Nesse modo, o tamanho total do elemento é calculado levando em consideração o conteúdo, as bordas e o preenchimento. Isso significa que, se você definir a largura de um elemento como 200 pixels, as bordas e o preenchimento serão incluídos nessa largura, resultando em um tamanho total menor. O conteúdo do elemento será dimensionado para se ajustar ao tamanho total especificado.

A propriedade box-sizing é útil quando você precisa controlar o tamanho total de um elemento de forma mais precisa, especialmente quando está trabalhando com layouts responsivos ou caixas de conteúdo com bordas e preenchimento.

Propriedade Font-family

A propriedade CSS **font-family** é usada para definir a família de fontes que será aplicada ao texto de um elemento HTML. Ela permite especificar uma lista de fontes preferenciais, separadas por vírgulas, para que o navegador tente usar a primeira fonte disponível na lista.

qui estão alguns pontos importantes a serem considerados sobre a propriedade font-family:

- A lista de fontes deve ser fornecida em ordem de preferência. O navegador tentará usar a primeira fonte da lista e, se não estiver disponível, passará para a próxima fonte e assim por diante.
- As fontes podem ser especificadas entre aspas, especialmente se o nome da fonte contiver espaços ou caracteres especiais. Por exemplo: font-family: "Arial", "Helvetica Neue", sans-serif;
- É uma boa prática fornecer uma família de fontes genérica no final da lista, como sans-serif, serif ou monospace. Isso garante que, se nenhuma das fontes preferenciais estiver disponível, o navegador usará uma fonte genérica adequada.
- É possível especificar várias fontes com o mesmo nome, mas em formatos diferentes. Por exemplo: font-family: Arial, Arial Black, sans-serif;. Nesse caso, o navegador tentará usar a primeira fonte disponível na lista, mas se ela não estiver disponível, passará para a próxima fonte.

A propriedade font-family é muito útil para definir a aparência do texto em um site. Ela permite que você escolha a fonte que melhor se adequa ao estilo e ao design do seu projeto.

Display: flex;

A **propriedade display: flex** em CSS é utilizada para criar layouts flexíveis, permitindo o alinhamento e distribuição de espaços entre os itens em um container. Essa propriedade é especialmente útil quando as dimensões dos itens são desconhecidas ou dinâmicas. Ao aplicar **display: flex** a um elemento, ele se torna um container flex e seus elementos filhos se tornam flex items. Para criar um container flex, basta adicionar a regra **display: flex** a um elemento.

Justify-content

A propriedade CSS justify-content é usada para alinhar os itens de um contêiner flexível ao longo do eixo principal. Ela define como o espaço disponível é distribuído entre os itens em relação ao eixo principal.

Existem várias opções para a propriedade justify-content:

- **flex-start** (valor padrão): Os itens são alinhados no início do contêiner flexível. Eles são empurrados para o início do eixo principal.
- **flex-end**: Os itens são alinhados no final do contêiner flexível. Eles são empurrados para o final do eixo principal.
- **center**: Os itens são centralizados ao longo do eixo principal do contêiner flexível.
- **space-between**: Os itens são distribuídos igualmente ao longo do eixo principal, com espaços em branco entre eles. O primeiro item é alinhado no início do contêiner e o último item é alinhado no final do contêiner.
- **space-around**: Os itens são distribuídos igualmente ao longo do eixo principal, com espaços em branco ao redor deles. Isso significa que há espaços iguais antes do primeiro item e depois do último item, e espaços iguais entre os itens.
- **space-evenly**: Os itens são distribuídos igualmente ao longo do eixo principal, com espaços em branco iguais entre eles e nos extremos do contêiner.

A propriedade justify-content é frequentemente usada em combinação com o contêiner flexível (display: flex) e a propriedade flex-direction para controlar o layout de um grupo de itens em um contêiner.

Elemento section

A **tag <section>** em HTML é usada para agrupar e organizar conteúdo relacionado em uma página da web. Ela representa uma seção tematicamente coesa de conteúdo dentro de um documento HTML.

A **tag <section>** é usada para dividir o conteúdo de uma página em partes distintas e significativas. Essas partes podem ser capítulos, tópicos, artigos, widgets ou qualquer outra divisão lógica do conteúdo.

A principal finalidade da **tag <section>** é melhorar a estrutura e a semântica do documento HTML, tornando-o mais legível e acessível para os usuários e mecanismos de busca. Ela também pode ser útil para estilizar e manipular o conteúdo usando CSS ou JavaScript.

A **tag <section>** não implica nenhuma formatação específica por si só. Ela é uma tag de contêiner genérica que pode ser estilizada e personalizada de acordo com as necessidades do desenvolvedor.

Vamos criar um section que engloba os nossos elementos do header.

```
14  <body>
15  <header>
16  |   <h2><i class="bi bi-film"></i>MyMovies</h2>
17  |   <section id="search-container">
18  |   |   <div class="input-wrapper">
19  |   |   |   <label for="movie-name">Nome:</label>
20  |   |   |   <input type="text" id="movie-name" />
21  |   |   </div>
22  |   |   <div class="input-wrapper">
23  |   |   |   <label for="movie-year">Ano:</label>
24  |   |   |   <input type="text" id="movie-year" />
25  |   |   </div>
26  |   |   <button><i class="bi bi-search"></i></button>
27  |   </section>
28  </header>
29  <main></main>
30  </body>
```



Propriedade Align-itens

A propriedade CSS align-items é usada para alinhar os itens de um contêiner flexível ao longo do eixo transversal. Ela define como os itens são posicionados verticalmente dentro do contêiner flexível.

Existem várias opções para a propriedade align-items:

- **stretch** (valor padrão): Os itens são esticados para preencher todo o espaço disponível no eixo transversal. Isso significa que eles terão a mesma altura do contêiner, a menos que sejam especificadas alturas fixas.
- **flex-start**: Os itens são alinhados no início do contêiner flexível ao longo do eixo transversal. Eles são empurrados para o início do eixo.
- **flex-end**: Os itens são alinhados no final do contêiner flexível ao longo do eixo transversal. Eles são empurrados para o final do eixo.
- **center**: Os itens são centralizados verticalmente ao longo do eixo transversal do contêiner flexível.
- **baseline**: Os itens são alinhados pela linha de base, que é a linha imaginária onde o texto é alinhado. Isso é útil quando você tem itens com alturas diferentes, mas deseja alinhá-los com base em seu texto.

A propriedade align-items é frequentemente usada em combinação com o contêiner flexível (display: flex) e a propriedade flex-direction para controlar o layout de um grupo de itens em um contêiner.

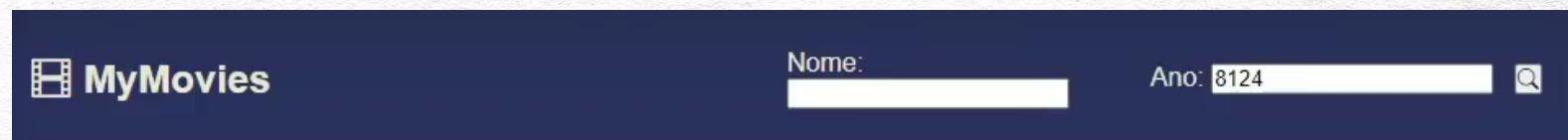
```
16 header {  
17   display: flex;  
18   justify-content: space-between;  
19   align-items: center;  
20   padding: 1.5rem 1rem;  
21   background-color: var(--primary-color);  
22 }  
23  
24 #search-container {  
25   display: flex;  
26   align-items: center;  
27 }
```



Módulo 14 – Estilizando o Header (7 / 8)

Nesse momento iremos fazer mais alguns ajustes de estilização do nosso Header, como cor, espaçamentos, tamanho dos elementos:

```
15
16 header {
17   display: flex;
18   justify-content: space-between;
19   align-items: center;
20   padding: 1.5rem 1rem;
21   background-color: var(--primary-color);
22   color: var(--lighest-color);
23 }
24
25 #search-container {
26   display: flex;
27   align-items: center;
28   width: 50%;
29 }
30
31 header > h2 > i {
32   margin-right: 0.5rem;
33 }
34
35 .input-wrapper {
36   width: 50%
37 }
```



Para melhorarmos ainda mais e chegar mais próximo da estilização desejada, iremos adicionar alguns seletores no nosso arquivo HTML para conseguir estilizar de maneira individualizada os elementos.

```
14 <body>
15   <header>
16     <h2><i class="bi bi-film"></i>MyMovies</h2>
17     <section id="search-container">
18       <div class="input-wrapper movie-name-wrapper">
19         <label for="movie-name">Nome:</label>
20         <input type="text" id="movie-name" />
21       </div>
22       <div class="input-wrapper movie-year-wrapper">
23         <label for="movie-year">Ano:</label>
24         <input type="text" id="movie-year" />
25       </div>
26       <button><i class="bi bi-search"></i></button>
27     </section>
28   </header>
29   <main></main>
30 </body>
```

```
31 header > h2 > i {
32   margin-right: 0.5rem;
33 }
34
35 .movie-name-wrapper {
36   width: 65%;
37 }
38
39 .movie-year-wrapper {
40   width: 30%;
41 }
42
43 #search-button {
44   width: 5%;
45 }
46
47 .movie-year-wrapper > input {
48   max-width: 5rem;
49 }
50
```



Propriedade flex-wrap

A propriedade "**flex-wrap**" é usada em CSS para controlar como os itens de um contêiner flexível são dispostos em uma única linha ou em várias linhas. Ela define se os itens devem ser ajustados em uma única linha (**valor "nowrap"**) ou se podem ser quebrados em várias linhas (**valor "wrap"**).

Quando o valor é definido como "**nowrap**", os itens são dispostos em uma única linha, mesmo que isso cause o estouro do contêiner. Por outro lado, quando o valor é definido como "**wrap**", os itens podem ser quebrados em várias linhas para se ajustarem ao contêiner.

Além disso, a propriedade "**flex-wrap**" também aceita o **valor "wrap-reverse"**, que quebra os itens em várias linhas, mas na ordem inversa. Ou seja, os itens são dispostos de trás para frente em relação à direção do fluxo principal.

É importante mencionar que a propriedade "**flex-wrap**" é aplicada ao contêiner flexível (elemento pai) e não aos itens individuais dentro dele. Ela trabalha em conjunto com outras propriedades flexíveis, como "flex-direction" e "justify-content", para controlar o layout dos itens flexíveis.

```
35 .movie-name-wrapper {  
36   width: 65%;  
37   display: flex;  
38   flex-wrap: nowrap;  
39 }
```



Módulo 14 – Os elementos de busca (2 / 5)

Vamos continuar estilizando os nossos elementos de busca (os inputs), para isso vamos adicionar padding, margin e algumas outras propriedades de CSS que já conhecemos os conceitos.

Propriedade font-weight

A propriedade "**font-weight**" é usada em CSS para definir a espessura ou o peso da fonte de um elemento de texto. Ela permite controlar a aparência visual do texto, tornando-o mais fino ou mais grosso.

A propriedade "**font-weight**" aceita diferentes valores numéricos ou palavras-chave para especificar o peso da fonte. Os valores numéricos variam de 100 a 900, sendo que 400 é o valor padrão e representa a espessura normal da fonte. Valores menores que 400 deixam a fonte mais fina, enquanto valores maiores que 400 a deixam mais grossa.

Além dos valores numéricos, também é possível utilizar palavras-chave para definir o peso da fonte. As palavras-chave mais comuns são:

- "**normal
- "**bold
- "**bolder
- "**lighter********

É importante mencionar que nem todas as fontes possuem variações de peso disponíveis. Portanto, o efeito da propriedade "font-weight" pode variar dependendo da fonte utilizada.

```

34
35 .movie-name-wrapper {
36   width: 65%;
37   display: flex;
38   align-items: center;
39 }
40
41 .movie-year-wrapper {
42   width: 30%;
43   display: flex;
44   align-items: center;
45 }
46
47 #search-button {
48   width: 5%;
49 }
50
51 .movie-year-wrapper > input {
52   max-width: 5rem;
53 }
54
55 .movie-name-wrapper > input {
56   width: 100%;
57 }
58
59 .movie-name-wrapper > input,
60 .movie-year-wrapper > input {
61   margin: 0 0.5rem;
62   border-radius: 7px;
63   padding: 0.3rem 0.2rem;
64   font-weight: bold;
65 }
66

```



Até esse momento a estilização do Header está com essa aparência:



Mas ainda temos algumas modificações para realizar até chegarmos no modelo desejado. Então vamos continuar com o nosso código.

Propriedade **all**

A propriedade "**all**" é usada em CSS para aplicar uma regra a todas as propriedades de um elemento de uma só vez. Ela permite que você defina um valor para todas as propriedades CSS de um elemento de forma rápida e conveniente.

Quando você define a propriedade "**all**" em um seletor CSS, ela afeta todas as propriedades do elemento, incluindo aquelas que já foram definidas anteriormente. Isso significa que qualquer valor definido anteriormente será substituído pelo novo valor especificado na propriedade "**all**".

O valor da propriedade "**all**" pode ser:

- "**initial- "**inherit- "**unset******

É importante ter cuidado ao usar a propriedade "**all**", pois ela pode ter um impacto significativo em todo o estilo do elemento e de seus elementos filhos. Ela deve ser usada com moderação e apenas quando necessário.

Além disso, é importante mencionar que a propriedade "**all**" não afeta as propriedades de animação, transição e alguns outros casos especiais. Essas propriedades continuam a ser definidas separadamente, mesmo se a propriedade "**all**" for usada.

Pseudo-classe :focus

A **pseudo-classe ":focus"** é usada em CSS para selecionar um elemento quando ele está em foco, ou seja, quando ele é o elemento ativo que está recebendo a atenção do usuário. Isso geralmente ocorre quando o usuário interage com o elemento usando o teclado ou o mouse.

Quando um elemento está em foco, é possível aplicar estilos específicos a ele para destacá-lo visualmente e fornecer feedback ao usuário. Por exemplo, você pode alterar a cor de fundo, a cor do texto ou adicionar uma borda ao elemento em foco.

A **pseudo-classe ":focus"** é especialmente útil para melhorar a usabilidade e a acessibilidade de um site ou aplicativo, pois ajuda os usuários a entenderem qual elemento está ativo e pronto para receber interações.

É importante mencionar que a **pseudo-classe ":focus"** só se aplica a elementos que podem receber foco, como campos de entrada de texto, botões e links. Além disso, ela não se aplica a elementos que não são interativos, como parágrafos ou divs.

Pseudo-classe: hover

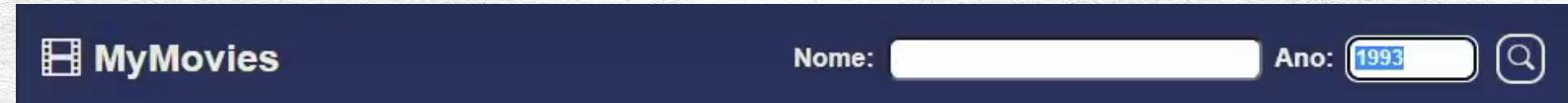
A **pseudo-classe "hover"** é usada em CSS para selecionar um elemento quando o cursor do mouse está sobre ele. Essa pseudo-classe permite aplicar estilos específicos a um elemento quando ele está sendo "hovered" (sobrevoado) pelo cursor do mouse.

Quando um elemento está em estado de "**:hover**", é possível alterar sua aparência visual para fornecer feedback ao usuário ou destacar o elemento interativo. Por exemplo, você pode alterar a cor de fundo, a cor do texto, adicionar uma borda ou qualquer outro estilo desejado.

A **pseudo-classe "hover"** é amplamente utilizada para melhorar a interatividade e a usabilidade de um site ou aplicativo. Ela permite que você crie efeitos visuais sutis que respondem às ações do usuário, tornando a experiência mais envolvente.

É importante mencionar que a **pseudo-classe "hover"** só se aplica a elementos que podem ser interativos, como botões, links e elementos de formulário. Ela não se aplica a elementos não interativos, como parágrafos ou divs.

```
66  
67 .movie-name-wrapper > label,  
68 .movie-year-wrapper > label {  
69   font-weight: bold;  
70 }  
71  
72 #search-button {  
73   all: unset;  
74   padding: 0.3rem;  
75   border: 2px solid var(--lighest-color);  
76   border-radius: 10px;  
77 }  
78  
79 #search-button:focus,  
80 #search-button:hover {  
81   background-color: var(--light-primary);  
82 }  
83
```



Nesse momento iremos realizar o último ajuste do nosso Header para adaptar para display maiores ou menores.

```
24  
25 #search-container {  
26   display: flex;  
27   align-items: center;  
28   width: 50%;  
29   justify-content: flex-end;  
30 }  
31  
32 header > h2 > i {  
33   margin-right: 0.5rem;  
34 }  
35  
36 .movie-name-wrapper {  
37   width: 65%;  
38   display: flex;  
39   align-items: center;  
40 }  
41  
42 .movie-year-wrapper {  
43   /* width: 30%; */  
44   display: flex;  
45   align-items: center;  
46 }
```



Continuaremos a trabalhar agora na nossa estrutura principal, ou seja, no nosso elemento main e adicionaremos tanto a sua estilização quanto os elementos desta estrutura.

```
29 <main>
30   <h2>Comece agora a sua lista de filmes!</h2>
31   <section id="movie-list"></section>
32 </main>
```



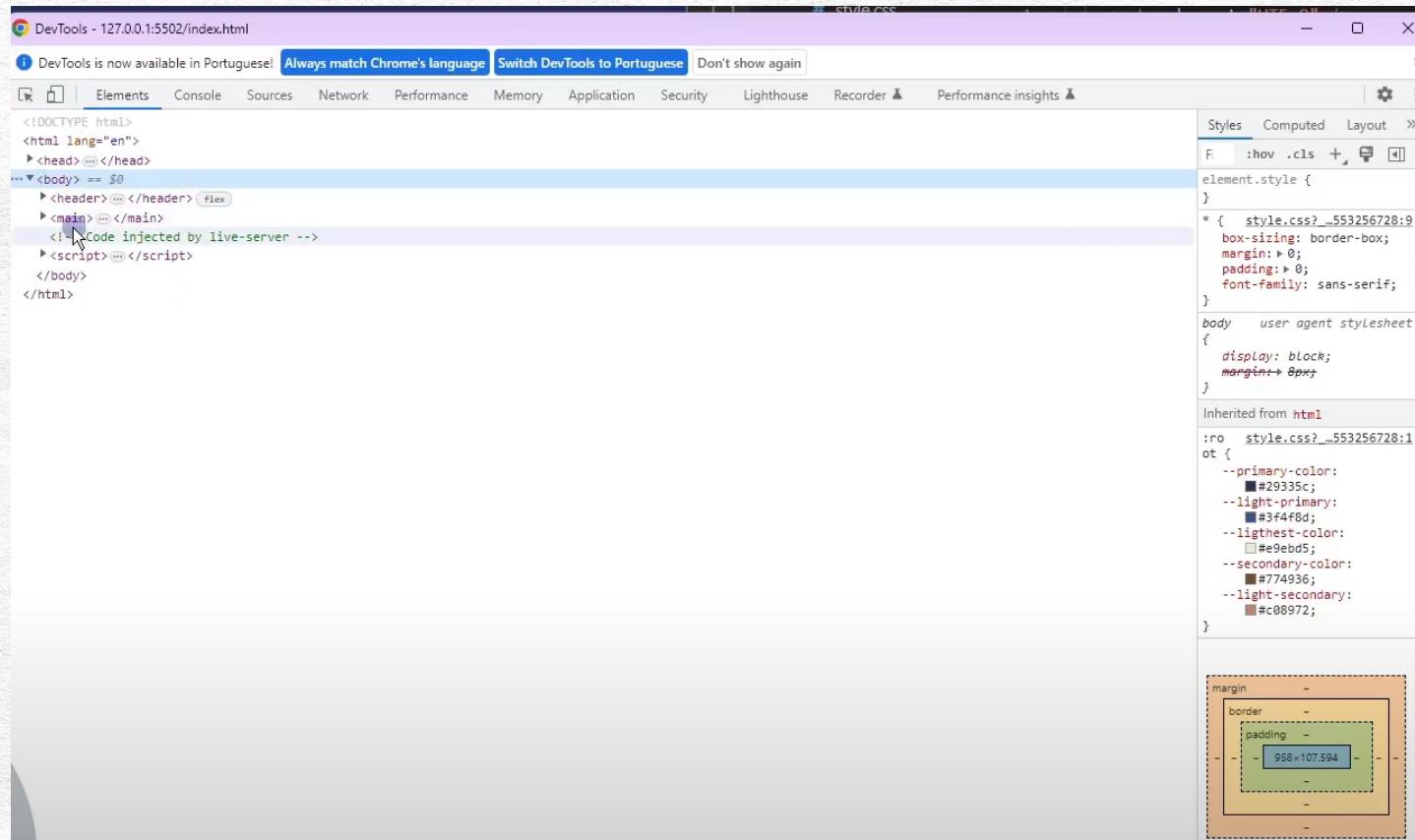
Estrutura de "pai" e "filho" no HTML

Em HTML, a relação pai e filho refere-se à hierarquia de elementos dentro da estrutura de marcação de uma página web. Essa relação é estabelecida quando um elemento HTML é colocado dentro de outro elemento HTML, criando uma relação de parentesco entre eles.

O elemento HTML que contém outro elemento é chamado de "pai" e o elemento contido é chamado de "filho". Essa relação é visualmente representada pela estrutura de árvore, onde o elemento pai é o nó superior e o elemento filho é um nó secundário. Essa relação de pai e filho é importante porque afeta a forma como os elementos são estilizados e manipulados por meio de CSS e JavaScript. Por exemplo, ao aplicar um estilo a um elemento pai, os elementos filhos podem herdar esses estilos, a menos que sejam especificamente sobreescritos.

Além disso, a relação de pai e filho também é relevante para a manipulação de elementos usando JavaScript. É possível acessar e modificar os elementos filhos por meio do elemento pai, usando métodos e propriedades do DOM (Document Object Model).

Você consegue visualizar essa relação pai e filho utilizando o DevTools (F12), lá você consegue observar quais elementos estão dentro de outros elementos e assim por diante.



Propriedade **flex-direction**

A propriedade "**flex-direction**" em CSS é usada para definir a direção em que os elementos flexíveis são exibidos dentro de um container flexível. Ela permite controlar a orientação dos itens em um layout flexível.

Existem quatro valores possíveis para a propriedade "**flex-direction**":

- "**row**" (padrão): Os itens são dispostos em uma linha da esquerda para a direita. Isso significa que os itens são alinhados horizontalmente.
- "**row-reverse**": Os itens são dispostos em uma linha da direita para a esquerda. A ordem dos itens é invertida em relação à ordem original.
- "**column**": Os itens são dispostos em uma coluna de cima para baixo. Isso significa que os itens são alinhados verticalmente.
- "**column-reverse**": Os itens são dispostos em uma coluna de baixo para cima. A ordem dos itens é invertida em relação à ordem original.

A propriedade "**flex-direction**" é especialmente útil quando se deseja criar layouts flexíveis e responsivos. Ela permite que você controle a direção em que os itens são exibidos, facilitando a adaptação do layout a diferentes tamanhos de tela e dispositivos.

Propriedade min-height

A propriedade "**min-height**" em CSS é usada para definir a altura mínima de um elemento. Ela especifica o valor mínimo que a altura do elemento pode ter, independentemente do conteúdo ou de outras propriedades que possam afetar a altura.

A propriedade "**min-height**" é útil quando você deseja garantir que um elemento tenha uma altura mínima específica, mesmo que o conteúdo dentro dele seja menor. Isso é especialmente útil em layouts responsivos, onde você deseja que os elementos se ajustem a diferentes tamanhos de tela, mas ainda tenham uma altura mínima consistente.

É importante mencionar que a propriedade "**min-height**" não limita a altura do elemento. Se o conteúdo dentro do elemento for maior do que a altura mínima especificada, o elemento será expandido para acomodar o conteúdo.

```
15  
16  body {  
17    display: flex;  
18    flex-direction: column;  
19    min-height: 100dvh;  
20  }  
21
```

A estilização do nosso main terá esse formato:

```
91  main {
92    text-align: center;
93    padding: 2rem 0;
94    background-color: var(--lighest-color);
95    color: var(--primary-color);
96    flex-grow: 1;
97  }
98
```



Elemento Article

A **tag <article>** é uma tag HTML que é usada para marcar um conteúdo independente e autônomo em uma página da web. Ela é usada para representar uma seção ou um bloco de conteúdo que é considerado completo por si só e pode ser distribuído ou reutilizado de forma independente em diferentes contextos.

A **tag <article>** é frequentemente usada para envolver conteúdo como notícias, postagens de blog, artigos, histórias, comentários, entre outros. Ela ajuda a organizar e estruturar o conteúdo da página, tornando mais fácil para os mecanismos de busca e leitores de tela entenderem a finalidade e o significado do conteúdo.

Ao usar a **tag <article>**, é importante garantir que o conteúdo dentro dela seja autônomo e tenha um significado completo, mesmo se for removido do contexto da página. Isso significa que o conteúdo dentro da **tag <article>** deve ser autoexplicativo e não depender de outros elementos da página para ser compreendido.

Além disso, a **tag <article>** também pode ser usada em conjunto com outras tags semânticas, como `<header>`, `<footer>`, `<section>`, para fornecer uma estrutura mais clara e significativa para o conteúdo da página.

A estrutura HMTL do nosso modal será criada com os elementos do código abaixo:

```
30  <main>
31      <h2>Comece agora a sua lista de filmes!</h2>
32      <section id="movie-list"></section>
33      <article id="modal-overlay">
34          <section id="modal-background"></section>
35          <section id="modal-container">
36              | <h2>Título do Filme</h2>
37              | </section>|
38          </article>
39      </main>
```

Agora, vamos começar a organizar o CSS para estilizar o nosso Modal. Para isso, vamos criar um arquivo chamado modal.css. Esse arquivo será responsável por conter todas as estilizações do nosso Modal. Essa separação é uma boa prática, pois nos permite ter estilos mais específicos em arquivos separados.

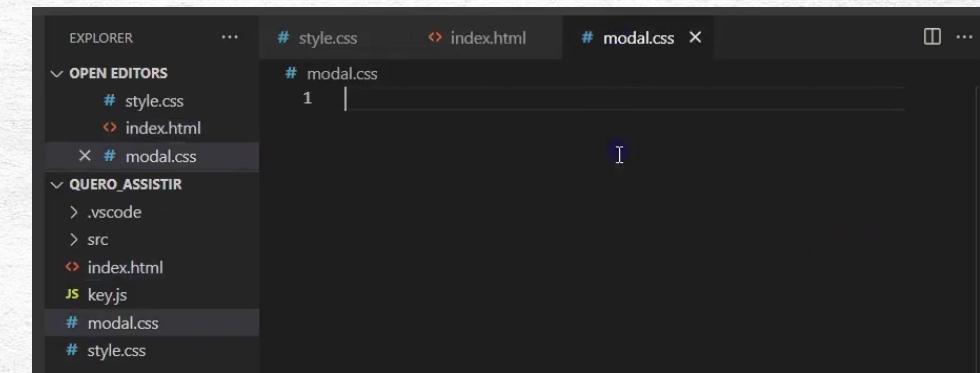
No nosso projeto, já temos um arquivo chamado style.css, que contém a estilização base para todo o projeto. No entanto, é comum termos estilos específicos para determinadas partes do nosso site ou aplicação. Ao separar esses estilos em arquivos diferentes, tornamos o projeto mais fácil de ser mantido, corrigido ou atualizado.

Dessa forma, no arquivo modal.css, iremos adicionar todas as estilizações necessárias para o nosso Modal. Podemos definir as cores, tamanhos, posicionamentos e outros estilos específicos para o Modal nesse arquivo. Assim, sempre que precisarmos fazer alterações ou adicionar novos estilos ao Modal, saberemos exatamente onde encontrar e modificar o código.

Essa prática de separar estilos específicos em arquivos diferentes nos ajuda a manter o projeto organizado e facilita a manutenção no futuro. Podemos atribuir estilos diferentes para cada parte do nosso projeto de forma mais clara e precisa.



```
3 <head>
4   <meta charset="UTF-8" />
5   <meta http-equiv="X-UA-Compatible" content="IE=edge" />
6   <meta name="viewport" content="width=device-width, ini
7   <title>Lista de filmes para assistir</title>
8   <link rel="stylesheet" href="./style.css" />
9   <link
10    rel="stylesheet"
11    href="https://cdn.jsdelivr.net/npm/bootstrap-icons@1.10.2/dist/
12    >
13    <link rel="stylesheet" href="./modal.css" />
14  </head>
```



Propriedade position

A **propriedade position** em CSS é usada para controlar o posicionamento de um elemento em relação ao seu contêiner ou à página como um todo. Ela permite que você defina como um elemento deve ser posicionado na página, levando em consideração outros elementos e o fluxo normal do documento.

Existem quatro valores principais que podem ser atribuídos à propriedade position:

- **static**: É o valor padrão. O elemento é posicionado de acordo com o fluxo normal do documento. Nenhuma propriedade de posicionamento adicional é aplicada.
- **relative**: O elemento é posicionado em relação à sua posição normal. Você pode usar as propriedades top, right, bottom e left para ajustar a posição do elemento em relação à sua posição normal. O espaço ocupado pelo elemento permanece reservado no layout normal da página.
- **absolute**: O elemento é posicionado em relação ao seu ancestral posicionado mais próximo. Se não houver nenhum ancestral posicionado, o elemento será posicionado em relação ao corpo do documento. O espaço ocupado pelo elemento é removido do fluxo normal do documento, o que significa que outros elementos podem ocupar seu lugar.
- **fixed**: O elemento é posicionado em relação à janela de visualização do navegador. Ele permanecerá fixo em sua posição, mesmo quando a página for rolada. O espaço ocupado pelo elemento também é removido do fluxo normal do documento.

Além desses valores, também existe o valor **sticky**, que combina características de relative e fixed. Um elemento com position: sticky é posicionado de acordo com o fluxo normal do documento até que seja rolado para fora da janela de visualização, momento em que se comporta como fixed.

A propriedade position é muito útil para criar layouts complexos e controlar o posicionamento dos elementos em uma página. No entanto, é importante usá-la com cuidado, levando em consideração a acessibilidade e a responsividade do design.

Propriedades top e left

As **propriedades top e left** são usadas em conjunto com a propriedade position em CSS para controlar o posicionamento de um elemento na página.

A **propriedade top** define a distância entre o elemento e o topo do seu contêiner ou do elemento pai posicionado. Você pode especificar essa distância usando valores absolutos, como pixels (por exemplo, top: 20px), ou valores relativos, como porcentagem (por exemplo, top: 50%). Um valor positivo move o elemento para baixo, enquanto um valor negativo move o elemento para cima.

A **propriedade left**, por sua vez, define a distância entre o elemento e a borda esquerda do seu contêiner ou do elemento pai posicionado. Assim como a propriedade top, você pode usar valores absolutos ou relativos para especificar essa distância. Um valor positivo move o elemento para a direita, enquanto um valor negativo move o elemento para a esquerda.

Essas propriedades são especialmente úteis quando o valor da propriedade position é definido como absolute ou fixed. Nesses casos, o elemento é removido do fluxo normal do documento e pode ser posicionado de forma precisa usando as propriedades top e left.

Aplicaremos agora no nosso modal as propriedades do CSS que acabamos de conhecer e algumas que já estamos familiarizados.

```
# modal.css > ...
1  #modal-overlay {
2    display: flex;
3    position: fixed;
4    top: 0;
5    left: 0;
6    width: 100vw;
7    height: 100dvh;
8    justify-content: center;
9    align-items: center;
10 }
11 |
```

Vamos estilizar o próximo elemento do nosso modal e conhecer novas propriedades do CSS.

Propriedade z-index

A **propriedade z-index** em CSS é usada para controlar a ordem de empilhamento dos elementos em uma página. Ela determina qual elemento deve aparecer na frente ou atrás de outros elementos, com base em um valor numérico.

O valor atribuído à **propriedade z-index** define a posição do elemento na pilha de camadas. Quanto maior o valor, mais acima o elemento será posicionado. Por exemplo, um elemento com z-index: 2 será colocado acima de um elemento com z-index: 1.

No entanto, é importante ressaltar que a propriedade z-index só tem efeito em elementos que possuem um valor de posicionamento diferente de static.

Isso significa que você precisa definir a propriedade position como relative, absolute ou fixed para que o z-index funcione corretamente.

Além disso, a **propriedade z-index** só afeta elementos que estão no mesmo contexto de empilhamento. Isso significa que, se um elemento estiver dentro de outro elemento com um contexto de empilhamento diferente, o z-index não terá efeito entre eles. Por exemplo, se você tiver dois elementos com position: relative e z-index: 1, mas um deles estiver dentro do outro, o elemento interno não será colocado acima do elemento externo, mesmo que tenha um valor de z-index maior.

A **propriedade z-index** é útil para controlar a sobreposição de elementos em um layout, especialmente quando há sobreposição de elementos posicionados. Ela permite que você crie camadas e controle a ordem em que os elementos são exibidos na página. No entanto, é importante usar o z-index com cuidado, pois um uso excessivo ou inadequado pode levar a problemas de usabilidade e acessibilidade. Certifique-se de testar e verificar como o z-index afeta a interação e a legibilidade do seu conteúdo.

Propriedade opacity

A **propriedade opacity** em CSS é usada para controlar a transparência de um elemento. Ela define o nível de opacidade do elemento, permitindo que você torne um elemento mais transparente ou mais opaco.

O valor da **propriedade opacity** varia de 0 a 1, onde 0 significa totalmente transparente e 1 significa totalmente opaco. Valores entre 0 e 1 representam diferentes níveis de transparência. Por exemplo, um valor de 0.5 torna o elemento 50% transparente, enquanto um valor de 0.75 torna o elemento 75% transparente.

Ao definir a **propriedade opacity** em um elemento, a transparência afeta não apenas o elemento em si, mas também todos os seus filhos. Isso significa que se você definir a opacidade de um elemento pai, todos os elementos filhos também terão a mesma opacidade.

É importante observar que a **propriedade opacity** afeta não apenas o conteúdo do elemento, mas também qualquer plano de fundo ou borda que o elemento possa ter. Isso significa que a transparência se aplica a todos os aspectos visuais do elemento. Além disso, a propriedade opacity também afeta a interação com o elemento. Quando um elemento tem uma opacidade menor que 1, ele se torna parcialmente transparente e os eventos do mouse não são mais direcionados a ele. Isso significa que o elemento não pode ser clicado ou interagido diretamente.

A **propriedade opacity** é útil para criar efeitos de transparência em elementos, permitindo que você crie sobreposições suaves, destaque elementos específicos ou crie efeitos visuais interessantes em seu design.

Módulo 14 – Estilizando o Modal (7 / 7)

Você construirá a estrutura do nosso arquivo modal.css como o código abaixo.

```
# modal.css > #modal-container
1  :root {
2    --primary-color: #29335c;
3    --light-primary: #3f4f8d;
4    --lighest-color: #e9ebd5;
5    --secondary-color: #774936;
6    --light-secondary: #c08972;
7  }
8
9  #modal-overlay {
10   display: flex;
11   position: fixed;
12   top: 0;
13   left: 0;
14   width: 100vw;
15   height: 100vh;
16   justify-content: center;
17   align-items: center;
18 }
19
20 #modal-background {
21   background-color: var(--light-primary);
22   width: 100%;
23   height: 100%;
24   position: absolute;
25   z-index: 0;
26   opacity: 0.4;
27 }
28
29 #modal-container {
30   z-index: 1;
31   background-color: var(--lighest-color);
32   padding: 1rem 3rem;
33   border-radius: 10px;
34 }
35
```



Nessa parte do projeto iremos montar um Mock de um Modal.

O que é Mock?

Em desenvolvimento de software, o **termo "mock"** se refere a uma técnica utilizada para simular o comportamento de um componente ou objeto real durante testes de software. Um "**mock**" é uma versão simulada de um objeto ou componente que é criada para imitar o comportamento do objeto real em um ambiente controlado.

O objetivo principal de usar **mocks** é isolar o código que está sendo testado, removendo dependências externas e garantindo que os testes sejam consistentes e previsíveis. Ao substituir componentes reais por mocks, os desenvolvedores podem simular diferentes cenários e comportamentos para verificar se o código está funcionando corretamente.

Os **mocks** são especialmente úteis quando se trata de testar interações com componentes externos, como bancos de dados, APIs ou serviços web. Em vez de depender desses componentes reais durante os testes, os mocks podem ser usados para fornecer respostas predefinidas ou simular comportamentos específicos.

Os **mocks** podem ser criados manualmente pelos desenvolvedores ou podem ser gerados automaticamente por ferramentas de teste específicas. Eles permitem que os desenvolvedores controlem o comportamento dos objetos simulados, definindo respostas, retornos de chamada e até mesmo simular erros ou exceções.

Ao usar **mocks**, os desenvolvedores podem testar o código de forma mais eficiente, identificar e corrigir problemas mais rapidamente e garantir que o software funcione corretamente em diferentes cenários.

Módulo 14 – Mock do primeiro Modal (2 / 6)

Para criarmos o nosso Mock do Modal, acessaremos o site oficial da API OMDB e pesquisaremos um exemplo de filme. A API retornará uma estrutura de arquivo json com a resposta da API.

JSON

JSON (JavaScript Object Notation) é um formato de dados amplamente utilizado em APIs (Interfaces de Programação de Aplicativos) para trocar informações entre um servidor e um cliente.

Quando uma API é chamada, ela retorna os dados em **formato JSON**. Esses dados são organizados em uma estrutura de pares de chave-valor, semelhante a um dicionário. Cada chave é uma string que identifica um determinado dado e cada valor pode ser um número, uma string, um booleano, um objeto ou um array.

O **JSON** é fácil de ser lido e interpretado por humanos, além de ser facilmente processado por máquinas. Ele é uma forma eficiente de transmitir dados entre diferentes sistemas, pois é leve e compacto.

Ao receber os dados em formato **JSON de uma API**, o cliente pode analisar esses dados e extrair as informações relevantes para serem exibidas na interface do usuário ou para serem usadas em outras partes do sistema.

The screenshot shows the OMDB API Examples page. At the top, there's a navigation bar with links for 'OMDb API', 'Usage', 'Parameters', 'Examples', 'Change Log', and 'API Key'. Below the navigation is a search bar with fields for 'Title' (set to 'Godfather'), 'Year' (set to '1972'), 'Plot' (set to 'Short'), and a dropdown for 'Response' (set to 'JSON'). There are 'Search' and 'Reset' buttons. Below the search form, there's a 'Request' section with the URL <http://www.omdbapi.com/?t=Godfather&y=1972>. The 'Response' section displays the JSON data for the movie 'The Godfather' from 1972.

```
{
    "Title": "The Godfather",
    "Year": "1972",
    "Rated": "R",
    "Released": "24 Mar 1972",
    "Runtime": "175 min",
    "Genre": "Crime, Drama",
    "Director": "Francis Ford Coppola",
    "Writer": "Mario Puzo, Francis Ford Coppola",
    "Actors": "Marlon Brando, Al Pacino, James Caan",
    "Plot": "The aging patriarch of an organized crime dynasty in postwar New York City transfers control of his clandestine empire to his reluctant youngest son.",
    "Language": "English, Italian, Latin",
    "Country": "United States",
    "Awards": "Won 3 Oscars. 32 wins & 31 nominations total",
    "Poster": "https://m.media-amazon.com/images/MV5BM2MyJlXXnimUtvTAwNj00MTYxLWjmNVYtzZODY3ZTK3OTTFkxEYkFqGdeQXyhIzkwMjQSNM@_V1_SX300.jpg",
    "Ratings": [
        {"Source": "Internet Movie Database", "Value": "9.2/10"},
        {"Source": "Rotten Tomatoes", "Value": "97%"},
        {"Source": "Metacritic", "Value": "100/100"}
    ],
    "Metascore": "100",
    "imdbRating": "9.2",
    "imdbVotes": "1,901,273",
    "imdbID": "tt0068646",
    "Type": "movie",
    "DVD": "11 May 2004",
    "BoxOffice": "$136,381,073",
    "Production": "N/A",
    "Website": "N/A",
    "Response": "True"
}
```

```
1  {
2      "Title": "The Godfather",
3      "Year": "1972",
4      "Rated": "R",
5      "Released": "24 Mar 1972",
6      "Runtime": "175 min",
7      "Genre": "Crime, Drama",
8      "Director": "Francis Ford Coppola",
9      "Writer": "Mario Puzo, Francis Ford Coppola",
10     "Actors": "Marlon Brando, Al Pacino, James Caan",
11     "Plot": "The aging patriarch of an organized crime dynas",
12     "Language": "English, Italian, Latin",
13     "Country": "United States",
14     "Awards": "Won 3 Oscars. 32 wins & 31 nominations total",
15     "Poster": "https://m.media-amazon.com/images/M/MV5BM2MyI",
16     "Ratings": [
17         { "Source": "Internet Movie Database", "Value": "9.2/10" },
18         { "Source": "Rotten Tomatoes", "Value": "97%" },
19         { "Source": "Metacritic", "Value": "100/100" }
20     ],
21     "Metascore": "100",
22     "imdbRating": "9.2",
23     "imdbVotes": "1,901,273",
24     "imdbID": "tt0068646",
25     "Type": "movie",
26     "DVD": "11 May 2004",
27     "Poster": "https://m.media-amazon.com/images/M/MV5BM2MyI..."
```

Vamos criar um arquivo no VS Code e colar o exemplo da resposta da API que mencionamos anteriormente. Esse exemplo representa como os dados serão retornados quando fizermos uma requisição na nossa própria API. Esses dados serão usados para simular o filme que estamos buscando e adicionar a estilização ao nosso Modal.

É importante destacar que o retorno da nossa API será um objeto JSON. Dentro desse objeto, teremos várias chaves e valores que representam informações sobre o filme, como o nome, o ano de lançamento, a imagem e outros detalhes.

Ao utilizar esse objeto JSON, poderemos extrair essas informações e utilizá-las para adicionar a estilização ao nosso Modal. Por exemplo, podemos exibir o nome do filme em um título, mostrar a imagem do filme em um elemento de imagem e exibir o ano de lançamento em um texto. Dessa forma, o objeto JSON que recebemos da API será fundamental para criar a aparência e o conteúdo do nosso Modal, garantindo que ele seja personalizado de acordo com as informações do filme que estamos buscando.

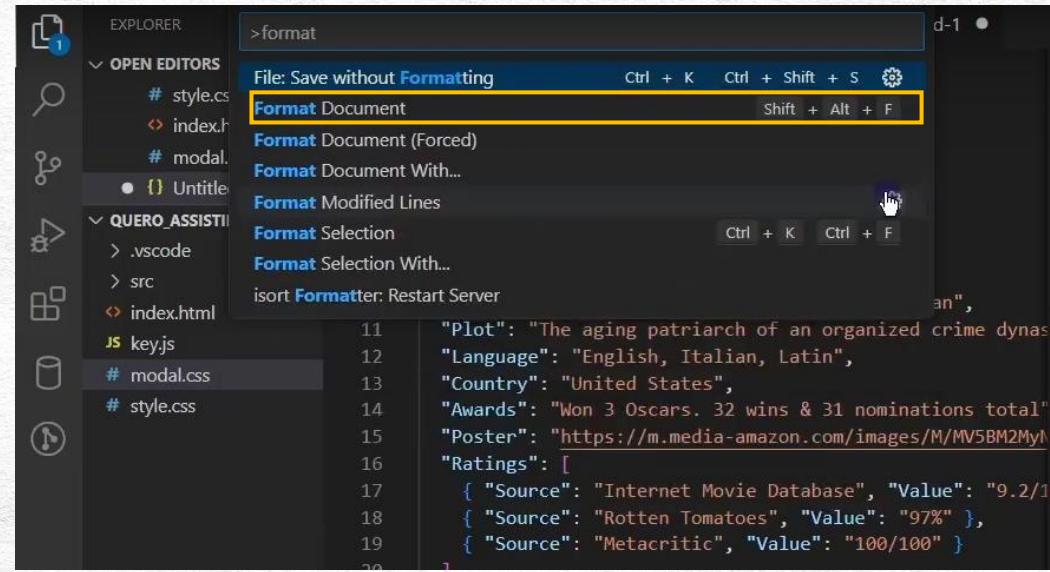
Em resumo, ao criar um arquivo no VS Code e colar o exemplo da resposta da API, estamos obtendo um objeto JSON que representa as informações do filme. Esse objeto será usado para adicionar a estilização e o conteúdo ao nosso Modal, garantindo que ele seja personalizado de acordo com os detalhes do filme que estamos buscando.



Format Document

O "Format Document" no VS Code é uma funcionalidade que permite formatar automaticamente o código-fonte de um arquivo de acordo com as regras de formatação definidas para a linguagem de programação em uso.

Você pode utilizar o atalho SHIFT + ALT + F.



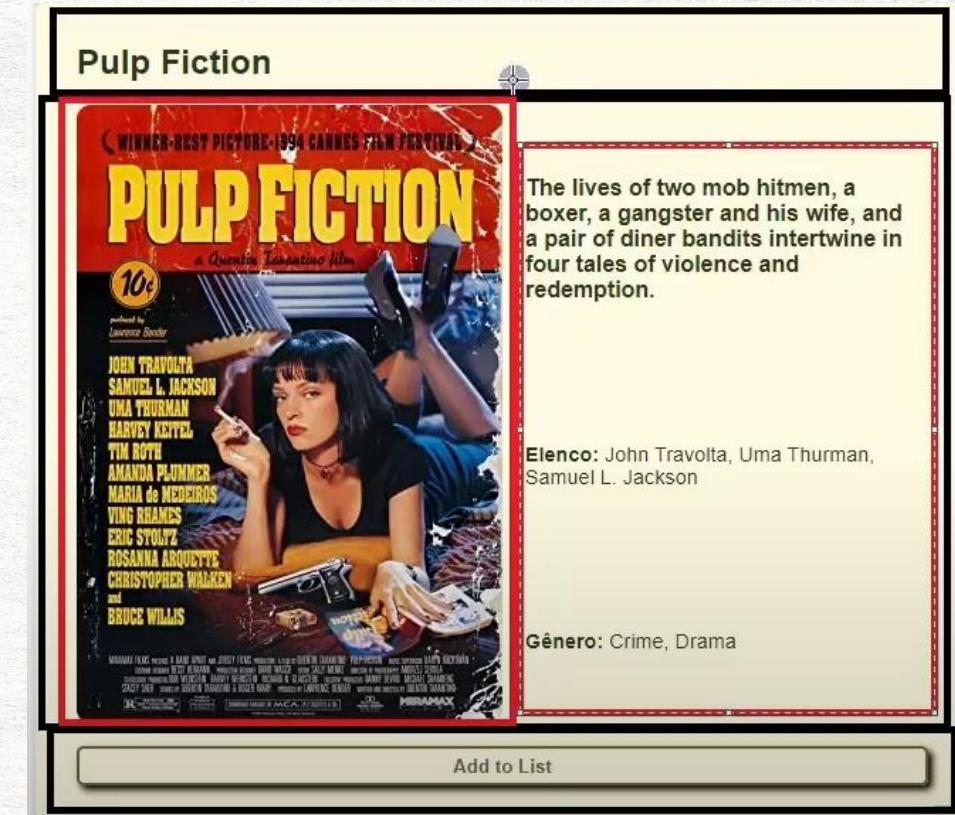
Ao analisarmos o nosso Modal (imagem ao lado), podemos dividir ele em três partes principais (cabeçalho, corpo e rodapé).

Dentro do corpo do nosso Modal, faremos uma subdivisão de duas partes, o local do poster e as informações do meu filme, como sinopse, elenco e gênero.

O cabeçalho possuirá o título do nosso filme, enquanto o rodapé possuirá o botão para adicionar o filme à nossa lista.

Conhecendo essa estrutura, vamos adicionar os elementos que faltam na estrutura do arquivo HTML do projeto:

```
30    <main>
31      <h2>Comece agora a sua lista de filmes!</h2>
32      <section id="movie-list"></section>
33      <article id="modal-overlay">
34          <section id="modal-background"></section>
35          <section id="modal-container">
36              <h2>Título do Filme</h2>
37              <section id="modal-body"></section>
38              <section id="modal-footer"></section>
39          </section>
40      </article>
41  </main>
```



Módulo 14 – Mock do primeiro Modal (6 / 6)

```
30  <main>
31    <h2>Comece agora a sua lista de filmes!</h2>
32    <section id="movie-list"></section>
33    <article id="modal-overlay">
34      <section id="modal-background"></section>
35      <section id="modal-container">
36        <h2>The Godfather - 1972</h2>
37        <section id="modal-body">
38          
42          <div id="movie-info">
43            <p id="movie-plot">
44              The aging patriarch of an organized crime
45              York City transfers control of his clandestine
46              reluctant youngest son.
47            </p>
48            <p id="movie-cast">Marlon Brando, Al Pacino,
49            <p id="movie-genre">Crime, Drama</p>
50          </div>
51        </section>
52        <section id="modal-footer">
53          <button id="add-to-list">Adicionar à Lista</button>
54        </section>
55      </section>
56    </article>
57  </main>
```

Criamos a estrutura principal dos elementos e vamos adicionar os elementos faltantes da parte principal, que são a imagem e as informações do filme.

E vamos colocar as informações que queremos de acordo com o JSON que tivemos como resposta da API, nesse momento ainda vamos deixar o Modal estático e depois iremos aplicar a inteligência a ele para conseguir estruturar qualquer filme que pesquisarmos.

Hierarquia do CSS

A hierarquia do CSS é a ordem em que as regras de estilo são aplicadas aos elementos HTML. Quando várias regras CSS se aplicam a um elemento, a hierarquia determina qual regra será usada para estilizar o elemento.

Aqui estão os principais conceitos da hierarquia do CSS:

- **Especificidade:** A especificidade é usada para determinar qual regra CSS é mais específica e, portanto, tem maior prioridade. Ela é baseada na combinação de seletores usados em uma regra CSS. Quanto mais específico for o seletor, maior será sua especificidade. Por exemplo, um seletor de classe tem maior especificidade do que um seletor de tag. Se houver conflito entre duas regras CSS, a regra com maior especificidade será aplicada.
- **Ordem:** A ordem em que as regras CSS são definidas também é importante na hierarquia do CSS. Se duas regras CSS tiverem a mesma especificidade, a regra que aparecer por último no código será aplicada. Portanto, é importante definir as regras CSS na ordem correta para garantir a aplicação correta dos estilos.
- **Importância:** Algumas declarações CSS têm uma importância maior do que outras. Por exemplo, declarações CSS definidas inline (no próprio elemento HTML) têm maior importância do que regras definidas em uma folha de estilos externa. Além disso, é possível usar a propriedade !important para dar uma importância maior a uma regra específica. No entanto, o uso excessivo de !important pode dificultar a manutenção do código e deve ser evitado.

Propriedade Cursor

A **propriedade cursor** no CSS é usada para definir o tipo de cursor que será exibido quando o mouse estiver sobre um elemento. Ela permite personalizar o cursor para indicar diferentes interações com o elemento, como clicar, passar o mouse ou arrastar.

Existem vários valores que podem ser atribuídos à propriedade cursor, cada um representando um tipo de cursor diferente. Alguns exemplos comuns são:

- **auto:** O navegador define o cursor padrão para o elemento.
- **pointer:** O cursor se transforma em uma mãozinha, indicando que o elemento é clicável.
- **default:** O cursor padrão do sistema operacional é usado.
- **text:** O cursor se transforma em um ícone de um cursor de texto, indicando que o elemento é editável.
- **move:** O cursor se transforma em uma seta com quatro pontas, indicando que o elemento pode ser arrastado.

Além desses, existem muitos outros valores que podem ser usados para personalizar o cursor de acordo com a interação desejada.

A propriedade cursor pode ser aplicada a qualquer elemento HTML e é uma forma simples de melhorar a usabilidade e a experiência do usuário em um site.

No arquivo style.css iremos aplicar algumas modificações que serão aplicadas a todos os botões do nosso projeto:

```
21  
22 button {  
23   all: unset;  
24   cursor: pointer;  
25 }
```

```
75  
76 #search-button {  
77   width: 5%;  
78   padding: 0.3rem;  
79   border: 2px solid var(--lighest-color);  
80   border-radius: 10px;  
81 }  
82  
83  
84  
85 }
```

Propriedade box-shadow

A **propriedade box-shadow** no CSS é usada para adicionar uma sombra a um elemento HTML. Ela permite criar efeitos de sombra personalizados para destacar elementos e dar profundidade ao design de uma página.

- **h-shadow**: Define o deslocamento horizontal da sombra. Um valor positivo move a sombra para a direita, enquanto um valor negativo move a sombra para a esquerda.
- **v-shadow**: Define o deslocamento vertical da sombra. Um valor positivo move a sombra para baixo, enquanto um valor negativo move a sombra para cima.
- **blur**: Define o desfoque da sombra. Quanto maior o valor, mais borrada será a sombra.
- **spread**: Define a expansão ou contração da sombra. Um valor positivo aumenta o tamanho da sombra, enquanto um valor negativo a reduz.
- **color**: Define a cor da sombra. Pode ser especificada usando um nome de cor, um valor hexadecimal ou uma função rgba().
- **inset (opcional)**: Define se a sombra é interna ou externa ao elemento. Se especificado como inset, a sombra será desenhada dentro do elemento.

Módulo 14 – CSS do Modal projetado (4 / 4)

Durante a criação da estilização de elementos de uma página web, é comum passarmos por um processo de tentativa e erro para alcançar o estilo desejado. Além de conhecer as propriedades do CSS, é importante testar diferentes estilos até encontrar o resultado desejado. Com o tempo e a prática, esse processo se torna mais fácil e intuitivo.

Então, para se aventurar no mundo do CSS, é interessante brincar com as propriedades e os valores que elas podem receber. Dessa forma, você poderá observar diferentes estilizações e descobrir como elas afetam os elementos da página. É uma ótima maneira de explorar e experimentar as possibilidades do CSS. Divirta-se e crie estilos únicos para os seus elementos!

Após algumas tentativas, finalizamos o nosso código do arquivo modal.css, que será usado para estilizar as partes principais do nosso modal. Dentro do seu projeto, o modal terá a aparência mostrada ao lado:

```

46
47 #movie-title {
48   padding: 1rem 0;
49 }
50
51 #modal-body {
52   display: flex;
53   justify-content: center;
54   padding: 1rem 0;
55 }
56
57 #movie-poster {
58   border-radius: 10px;
59 }
60
61 #movie-info {
62   display: flex;
63   flex-direction: column;
64   justify-content: space-around;
65   max-width: 300px;
66 }
67
68 #add-to-list {
69   border: 3px solid var(--primary-color);
70   padding: 0.3rem 0;
71   width: 100%;
72   border-radius: 10px;
73   font-weight: bold;
74   box-shadow: 1px 1px 10px var(--primary-color);
75 }
76
77 #add-to-list:focus,
78 #add-to-list:hover {
79   background-color: var(--light-primary);
80   color: var(--ligthest-color);
81 }
82

```



Para começar a implementar a lógica do nosso Modal, vamos criar um arquivo chamado modal.js e escrever o código JavaScript nele. Essa separação de responsabilidades nos ajudará a organizar melhor o nosso projeto.

Dentro do arquivo modal.js, poderemos adicionar funções e lógica de programação para controlar o comportamento do Modal. Por exemplo, podemos escrever código para abrir e fechar o Modal quando um botão for clicado, ou para validar os dados inseridos em um formulário dentro do Modal.

Essa abordagem de separar a lógica do Modal em um arquivo JavaScript específico nos permite manter o código mais organizado e facilita a manutenção e reutilização do código. Além disso, também nos permite ter um melhor controle sobre as interações e funcionalidades do Modal.

Portanto, ao criar o arquivo modal.js e escrever a lógica do nosso Modal nele, estaremos seguindo boas práticas de programação e tornando o nosso código mais modular e fácil de gerenciar.

E para iniciarmos a nossa lógica iremos conhecer alguns conceitos que iremos aplicar durante a construção do Javascript.

Document.getElementById

A função **document.getElementById** é uma função JavaScript que nos permite obter uma referência a um elemento HTML específico na página com base no seu atributo id.

- **id:** É uma string que representa o valor do atributo id do elemento HTML que queremos obter.

Ao chamar a função **document.getElementById** e passar o valor do id como argumento, a função retorna um objeto que representa o elemento HTML correspondente. Podemos armazenar esse objeto em uma variável para usá-lo posteriormente.

A função **document.getElementById** é muito útil quando precisamos interagir com elementos específicos em nossa página, como alterar seu conteúdo, estilo ou adicionar eventos. Ela nos permite acessar esses elementos de forma fácil e eficiente usando seus atributos id.

```
JS modal.js > ...
1 const background = document.getElementById('modal-background');
2 const modalContainer = document.getElementById('modal-container');
```

Para adicionar a funcionalidade de mostrar e ocultar o Modal, precisamos fazer alguns ajustes no nosso arquivo HTML e na folha de estilos CSS.

Arquivo HTML

```
33 <article id="modal-overlay" class="overlay">  
34 |<div id="modal-content"></div>  
35 |</article>  
36 |</body>  
37 |</html>  
38  
39 </main>  
40 <script src="./modal.js"></script>  
41 </body>  
42 </html>
```

Arquivo CSS – modal.css

```
20 .overlay {  
21 | visibility: hidden;  
22 }  
23  
24 .open {  
25 | visibility: visible;  
26 }
```

classList.add()

A **propriedade classList** em JavaScript é usada para acessar a lista de classes de um elemento HTML. Através dessa propriedade, podemos adicionar, remover ou verificar a presença de classes em um elemento.

A **função add** do classList é usada para adicionar uma classe a um elemento HTML.

Ao chamar **classList.add** e passar o nome da classe como argumento, a classe é adicionada à lista de classes do elemento. Se a classe já estiver presente, ela não será adicionada novamente.

A **função classList.add** é útil quando queremos adicionar ou modificar classes em elementos HTML dinamicamente, por exemplo, em resposta a eventos ou condições específicas. Ela nos permite manipular as classes de forma fácil e eficiente, ajudando a controlar o estilo e o comportamento dos elementos em nossa página.



AddEventListener

O **método addEventListener** em JavaScript é usado para adicionar um ouvinte de eventos a um elemento HTML. Ele permite que você especifique uma função que será executada quando um determinado evento ocorrer no elemento.

A sintaxe básica do **addEventListener** é a seguinte:

```
elemento.addEventListener(evento, função);
```

Ao chamar **addEventListener** e passar o tipo de evento e a função como argumentos, a função será associada ao elemento e será executada sempre que o evento especificado ocorrer.

O **addEventListener** é muito útil quando queremos adicionar interatividade aos elementos da nossa página. Podemos usar esse método para responder a eventos como cliques, movimentos do mouse, pressionamentos de teclas e muito mais. Ele nos permite criar comportamentos dinâmicos e interativos em nossos elementos HTML, tornando a experiência do usuário mais rica e envolvente.

classList.remove

A **função remove** do `classList` é usada para remover uma classe de um elemento HTML.

Ao chamar **classList.remove** e passar o nome da classe como argumento, a classe é removida da lista de classes do elemento. Se a classe não estiver presente, a função não faz nada.

A função **classList.remove** é útil quando queremos remover ou modificar classes em elementos HTML dinamicamente, por exemplo, em resposta a eventos ou condições específicas. Ela nos permite manipular as classes de forma fácil e eficiente, ajudando a controlar o estilo e o comportamento dos elementos em nossa página.

Propriedade Transition

A **propriedade transition** em CSS é usada para criar transições suaves entre diferentes estados de um elemento. Ela permite controlar a animação de mudanças em propriedades específicas, como cor, tamanho, posição, entre outras.

A sintaxe básica da propriedade transition é: **transition: propriedade duração tipo atraso;**

- **propriedade:** É a propriedade CSS que queremos animar durante a transição. Podemos especificar várias propriedades separadas por vírgula.
- **duração:** É o tempo que a transição deve levar para ser concluída. Podemos usar valores em segundos (s) ou milissegundos (ms).
- **tipo:** É o tipo de transição que queremos aplicar. Existem diferentes tipos disponíveis, como linear, ease, ease-in, ease-out, ease-in-out, entre outros.
- **atraso (opcional):** É o tempo que deve ser aguardado antes de iniciar a transição. Podemos usar valores em segundos (s) ou milissegundos (ms).

Propriedade Opacity

A **propriedade opacity** em CSS é usada para controlar a opacidade de um elemento HTML. Ela define o nível de transparência do elemento, permitindo que você torne um elemento mais ou menos visível.

A **propriedade opacity** aceita valores entre 0 e 1, onde 0 significa totalmente transparente (invisível) e 1 significa totalmente opaco (totalmente visível). Valores decimais entre 0 e 1 podem ser usados para definir diferentes níveis de transparência.

Vamos nesse momento adicionar os conceitos que acabamos de aprender nos arquivos modal.css e modal.js. A estilização e lógica de programação aplicada, serve para que ao clicarmos no elemento de busca nos motre o Modal e ao clicarmos forma ele desapareça.

Arquivo modal.css

```
20  .overlay {  
21    visibility: hidden;  
22    transition: all 0.3s ease;  
23    opacity: 0;  
24  }  
25  
26  .open {  
27    visibility: visible;  
28    opacity: 1;  
29  }
```

Arquivo modal.js

```
const background = document.getElementById('modal-background');  
const modalContainer = document.getElementById('modal-container');  
  
function backgroundClickHandler() {  
  overlay.classList.remove('open');  
}  
  
background.addEventListener('click', backgroundClickHandler);
```



Vamos começar a implementar a lógica de programação na página principal do nosso projeto. Para isso, iremos seguir os seguintes passos:

- Criar um arquivo chamado "script.js" para armazenar o código JavaScript.
- Importar o arquivo "script.js" para o nosso arquivo "index.html".
- Adicionar as funcionalidades que desejamos na página.

```
</main>
<script src="./script.js"></script>
<script src="./modal.js"></script>
<script src="./src/notie-master/dist/notie.min.js"></script>
</body>
</html>
```

```
const $searchButton = document.getElementById('search-button');
```

O símbolo "\$" antes do nome da variável é uma convenção comumente usada em muitos frameworks e bibliotecas JavaScript, como o jQuery. Essa convenção é conhecida como "alias" ou "apelido" e é usada para indicar que a variável está sendo usada para armazenar uma referência a um elemento do DOM (Documento Object Model) que foi selecionado usando uma função específica, como o "" do jQuery.

No exemplo acima, a variável "\$searchButton" está sendo usada para armazenar uma referência ao elemento do DOM com o ID "search-button". Essa referência pode ser usada posteriormente para manipular esse elemento, como adicionar um evento de clique ou alterar suas propriedades.

É importante observar que o uso do símbolo "\$" antes do nome da variável não é uma sintaxe padrão do JavaScript, mas sim uma convenção adotada por algumas bibliotecas e frameworks para facilitar a identificação de variáveis que representam elementos do DOM.

Vamos implementar a funcionalidade de fazer o Modal aparecer quando o botão de busca (ícone de lupa) for clicado. Para isso, siga o código abaixo:

```
JS script.js > ...
1  const searchButton = document.getElementById('search-button');
2  const overlay = document.getElementById('modal-overlay');
3
4  function searchButtonClickHandler() {
5    overlay.classList.add('open');
6  }
7
8  searchButton.addEventListener('click', searchButtonClickHandler);
9
```

Neste momento, vamos aprender como obter o retorno da nossa API. Para isso, vamos usar a URL [http://www.omdbapi.com/?apikey=\[yourkey\]&](http://www.omdbapi.com/?apikey=[yourkey]&) e definir quais parâmetros queremos retornar. No caso do nosso projeto, queremos obter o nome do filme e o ano. O ano é opcional e será usado para lidar com qualquer ambiguidade de filmes que possam existir.

Propriedade .value (Javascript)

.value é uma propriedade em JavaScript. Em JavaScript, as propriedades são características de um objeto que podem ser acessadas e modificadas. No caso de elementos de formulário, como campos de texto, caixas de seleção e botões de opção, a propriedade .value é usada para acessar ou definir o valor atual do elemento.

Por exemplo, se você tiver um elemento de formulário representado por uma variável chamada input, você pode usar input.value para acessar o valor atual desse elemento. Você também pode atribuir um novo valor a essa propriedade para atualizar o valor do elemento.

A **propriedade .value** é comumente usada para obter o valor de campos de entrada de texto, como <input type="text">, <textarea>, <select>, entre outros. No entanto, é importante observar que nem todos os elementos HTML têm a propriedade .value. Alguns elementos, como <div> ou , não possuem essa propriedade.

Método **split()**

A método **split()** em JavaScript é usada para dividir uma string em um array de substrings com base em um separador especificado.

Quando você chama a método **split()** em uma string, você fornece um separador como argumento. Esse separador pode ser um caractere, uma sequência de caracteres ou uma expressão regular. A string é então dividida em partes sempre que o separador é encontrado.

Método **join()**

O método **join()** em JavaScript é usado para juntar os elementos de um array em uma única string. Ele combina os elementos do array usando um separador especificado e retorna a string resultante.

Quando você chama o método **join()** em um array, você pode fornecer um separador como argumento. Esse separador será inserido entre cada par de elementos do array ao criar a string resultante.

Para entender como a nossa API retorna a URL do filme buscado, precisamos seguir alguns passos:

- Acesse o site da API que estamos utilizando.
- Faça uma pesquisa de um filme específico.
- Observe o formato em que a URL é retornada.
- Com base nesse formato, podemos aplicar a lógica de programação no nosso arquivo script.js.

Dessa forma, ao analisar a URL retornada pela API, podemos entender como extrair as informações necessárias para o nosso projeto e utilizar essas informações no nosso código JavaScript. Isso nos permitirá trabalhar com os dados do filme de forma adequada.

The screenshot shows the OMDb API Examples page. At the top, there is a navigation bar with links for 'OMDb API', 'Usage', 'Parameters', 'Examples', 'Change Log', and 'API Key'. To the right of the navigation bar are buttons for 'Become a Patron', 'Donate', and 'Contact'. Below the navigation bar, there is a section titled 'By Title' with a form. The 'Title' field contains 'Godfather Part II', the 'Year' field contains '1974', the 'Plot' dropdown is set to 'Short', and the 'Response' dropdown is set to 'JSON'. There are 'Search' and 'Reset' buttons. Below the form, under the heading 'Request:', is a link to the generated API URL: <http://www.omdbapi.com/?t=Godfather+Part+II&y=1974>. Under the heading 'Response:', the JSON data returned by the API is displayed in a green box. The JSON object contains details about the movie 'The Godfather Part II', including its title, year, cast, plot summary, and various ratings and awards.

```
{"Title": "The Godfather Part II", "Year": "1974", "Rated": "R", "Released": "18 Dec 1974", "Runtime": "202 min", "Genre": "Crime, Drama", "Director": "Francis Ford Coppola", "Writer": "Francis Ford Coppola, Mario Puzo", "Actors": "Al Pacino, Robert De Niro, Robert Duvall", "Plot": "The early life and career of Vito Corleone in 1920s New York City is portrayed, while his son, Michael, expands and tightens his grip on the family crime syndicate.", "Language": "English, Italian, Spanish, Latin, Sicilian", "Country": "United States", "Awards": "Won 6 Oscars. 17 wins & 21 nominations", "Poster": "https://m.media-amazon.com/images/M/MV5BMWMwMGQzZTIt2JlNC00OWZlWjyMDctNDkZDQ2YjRjMWQ0XkeyXkFqcGdeQXVjNzkwMjQ5NzM@_V1_SX300.jpg", "Ratings": [{"Source": "Internet Movie Database", "Value": "9.0/10"}, {"Source": "Rotten Tomatoes", "Value": "96%"}, {"Source": "Metacritic", "Value": "90/100"}], "Metascore": "90", "imdbRating": "9.0", "imdbVotes": "1,307,220", "imdbID": "tt0071562", "Type": "movie", "DVD": "24 May 2005", "BoxOffice": "$47,834,595", "Production": "N/A", "Website": "N/A", "Response": "True"}
```

Entendendo como que é o retorno da nossa API, vamos construir o nosso código em Javascript utilizando os métodos e o formato que nossa API retorna a URL.

```
js script.js > ⚙️ searchButtonClickHandler
1  const searchButton = document.getElementById('search-button');
2  const overlay = document.getElementById('modal-overlay');
3  const movieName = document.getElementById('movie-name');
4  const movieYear = document.getElementById('movie-year');
5
6  function searchButtonClickHandler() {
7    overlay.classList.add('open');
8    console.log(movieName.value.split(' ').join('+'));
9    console.log('Ano: ', movieYear.value);
10 }
11
12 searchButton.addEventListener('click', searchButtonClickHandler);
13
```

Requisição tipo GET

Uma requisição do tipo **GET**, em português "obter" ou "buscar", é um tipo de requisição feita em uma aplicação web para obter dados de um servidor.

Quando fazemos uma requisição **GET**, estamos solicitando ao servidor que nos forneça um recurso específico, como uma página da web, um arquivo de imagem ou um conjunto de dados. Essa solicitação é feita através de uma URL (Uniform Resource Locator) que identifica o recurso desejado.

Ao enviar uma requisição **GET**, não estamos enviando nenhum dado adicional ao servidor além da própria URL. A requisição é feita através do protocolo HTTP (Hypertext Transfer Protocol) e é considerada uma operação segura e idempotente, o que significa que não deve ter efeitos colaterais e pode ser repetida várias vezes sem alterar o estado do servidor.

Por exemplo, se quisermos obter informações sobre um filme de uma API, podemos fazer uma requisição GET para a URL específica do recurso desejado. O servidor então nos retornará os dados do filme, que podem ser no formato **JSON**, XML ou outro formato.

Examples

By Title

Title: Empire Strikes Back Year: 1980 Plot: Short Response:

Request:

<http://www.omdbapi.com/?t=Empire+Strikes+Back&y=1980>

Response:

```
{"Title": "Star Wars: Episode V - The Empire Strikes Back", "Year": "1980", "Rated": "PG", "Released": "20 Jun 1980", "Runtime": "124 min", "Genre": "Action, Adventure, Fantasy", "Director": "Irvin Kershner", "Writer": "Leigh Brackett, Lawrence Kasdan, George Lucas", "Actors": "Mark Hamill, Harrison Ford, Carrie Fisher", "Plot": "After the Rebels are over powered by the Empire, Luke Skywalker begins his Jedi training with Yoda, while his friends are pursued across the galaxy by Darth Vader and bounty hunter Boba Fett.", "Language": "English", "Country": "United States", "Awards": "Won 1 Oscar. 26 wins & 20 nominations total", "Poster": "https://m.media-amazon.com/images/M/MV5BYmU1NDRjNDgtMzhiMjQ0NjZmLTg5NGIzMjZDNlZjUSNTU4OTE0XkEyXkFqcGdeQXVyNzkwMjQ5NzMA._V1_SX300.jpg", "Ratings": [{"Source": "Internet Movie Database", "Value": "8.7/10"}, {"Source": "Rotten Tomatoes", "Value": "94%"}, {"Source": "Metacritic", "Value": "82/100"}], "Metascore": "82", "imdbRating": "8.7", "imdbVotes": "1,316,200", "imdbID": "tt0080684", "Type": "movie", "DVD": "21 Sep 2004", "BoxOffice": "$292,753,960", "Production": "N/A", "Website": "N/A", "Response": "True"}
```

Sabemos que o retorno da nossa API é no formato JSON e que as querys das URL tem um formato específico, que podemos obter através do site oficial da OMDB.

Observe que os parâmetros da nossa URL, vem com t=Titulo do Filme e y=Ano, então será com essas estruturas que iremos trabalhar.

Módulo 14 – Consumindo a API do OMDB (3 / 5)

Aplicaremos o inicio da lógica para verificarmos se as requisições que iremos fazer a nossa API irá retornar de maneira correta.

O código inicial será esse ao lado, e vamos fazer os seguintes testes:

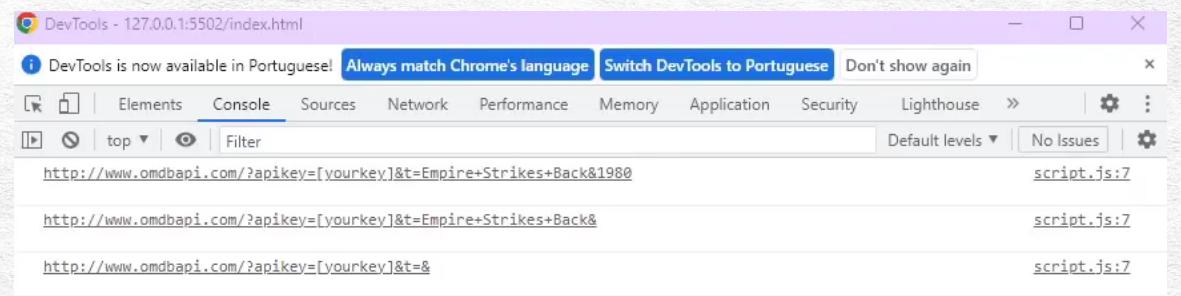
- Buscando por Título do Filme e Ano
- Buscando apenas com o Título do Filme
- Fazendo uma busca vazia

Repare que ele retorna as 3 requisições, porém quando não for passado o Ano, devemos corrigir o último "&", porque não teremos nada em seguida.

E se fizermos uma requisição vazia, concorda que não deveria retornar NADA? Afinal não existe uma requisição sendo feito.

Então vamos ter que ajustar o nosso código e corrigir esses problemas.

```
JS script.js > ...
1 const searchButton = document.getElementById('search-button');
2 const overlay = document.getElementById('modal-overlay');
3 const movieName = document.getElementById('movie-name');
4 const movieYear = document.getElementById('movie-year');
5
6 function searchButtonClickHandler() {
7   console.log(`http://www.omdbapi.com/?apikey=[yourkey]&t=${movieName.value
8     .split(' ')
9     .join('+')}&${movieYear.value}
10    `);
11   overlay.classList.add('open');
12   // console.log(movieName.value.split(' ').join('+'));
13   // console.log('Ano: ', movieYear.value);
14 }
15
16 searchButton.addEventListener('click', searchButtonClickHandler);
17
```



No começo do projeto criamos uma chave para utilizarmos na nossa API, essa chave foi feita através do cadastro e enviado por email. É nesse momento que utilizaremos ela, vamos criar um arquivo chamado **key.js**.

Dentro dele iremos armazenar uma variável e atribuir o valor dessa nossa chave (**const key = 'chave pessoal da API';**) , esse será o conteúdo do seu arquivo.

FETCH

O **fetch** é uma função em JavaScript que é usada para fazer requisições de recursos em uma rede, como uma API ou um arquivo. Ele é uma maneira moderna e mais flexível de fazer requisições HTTP em comparação com as abordagens mais antigas, como o uso do XMLHttpRequest.

Ao usar o **fetch**, você pode enviar uma requisição HTTP para um determinado URL e receber a resposta do servidor. Essa resposta pode ser um objeto Response que contém informações sobre a resposta, como o status da requisição, os cabeçalhos e o corpo da resposta.

O **fetch** retorna uma promessa, o que significa que você pode usar **os métodos .then() e .catch()** para lidar com a resposta da requisição. Você pode encadear várias chamadas .then() para manipular os dados retornados e realizar as ações desejadas.

Vamos fazer uma requisição GET para a URL 'da nossa API'. Em seguida, usamos o **método .json()** para extrair os dados da resposta como um objeto JSON. Em seguida, podemos manipular esses dados dentro do segundo **.then()**.

Se ocorrer algum erro durante a requisição, o **.catch()** será acionado e podemos lidar com o erro de acordo.

O fetch também permite configurar a requisição, como adicionar cabeçalhos personalizados, enviar dados no corpo da requisição e definir o método da requisição (GET, POST, PUT, DELETE, etc.).

ASYNC / AWAIT

O **async/await** é uma funcionalidade do JavaScript que permite escrever código assíncrono de forma mais síncrona e fácil de ler. Ele é baseado em promessas e fornece uma maneira mais intuitiva de lidar com operações assíncronas.

Quando uma função é declarada com a palavra-chave `async`, ela se torna uma função assíncrona. Dentro dessa função, podemos usar a palavra-chave `await` antes de uma expressão que retorna uma promessa. O `await` pausa a execução da função assíncrona até que a promessa seja resolvida ou rejeitada.

A principal vantagem do **async/await** é que ele permite escrever código assíncrono de forma sequencial, como se estivéssemos escrevendo código síncrono. Isso torna o código mais legível e mais fácil de entender.

Com esse novos conceitos conseguimos continuar a construir nosso código:

```
JS script.js  X
quero_assistir > JS script.js > ...
1  const searchButton = document.getElementById('search-button');
2  const overlay = document.getElementById('modal-overlay');
3  const movieName = document.getElementById('movie-name');
4  const movieYear = document.getElementById('movie-year');
5
6  async function searchButtonClickHandler() {
7    let url = `http://www.omdbapi.com/?apikey=${key}&t=${movieName.value.split(' ').join('+')}&${movieYear.value}`;
8    const response = await fetch(url);
9    const data = await response.json();
10   overlay.classList.add('open');
11 }
12
13 searchButton.addEventListener('click', searchButtonClickHandler);
14 |
```



TRY/CATCH

O **try-catch** é uma estrutura de controle em programação que permite capturar e tratar exceções que podem ocorrer durante a execução de um bloco de código. Com o try-catch, é possível prever possíveis erros e lidar com eles de forma adequada, evitando que o programa seja interrompido abruptamente.

- O bloco **try** contém o código que pode gerar uma exceção.
- O bloco **catch** é executado apenas se uma exceção do tipo especificado ocorrer dentro do bloco try.
- O parâmetro *e* é uma variável que armazena a exceção lançada, permitindo que você acesse informações sobre a exceção, como a mensagem de erro.

No caso de múltiplos blocos catch, eles são verificados em ordem e o primeiro bloco cujo tipo de exceção corresponda ao lançado será executado. Se nenhuma exceção for lançada dentro do bloco try, o bloco catch será ignorado e a execução continuará normalmente.

É importante ressaltar que a **estrutura try-catch** deve ser usada com cuidado e apenas quando necessário. É recomendado capturar exceções específicas e tratar cada tipo de exceção de forma adequada, em vez de capturar todas as exceções em um único bloco catch.

```
6  async function searchButtonClickHandler() {
7    try {
8      let url = `http://www.omdbapi.com/?apikey=${key}&t=${movieName.value}
9        .split(' ')
10       .join('+')&y=${movieYear.value}
11     `;
12     const response = await fetch(url);
13     const data = await response.json();
14     console.log('data: ', data);
15     overlay.classList.add('open');      You, last week • initial commit
16   } catch (error) {
17   }
18 }
19 }
```



Separar funções para diferentes tarefas pode ajudar a melhorar o desempenho do seu código. Funções menores e mais específicas tendem a ser mais eficientes do que funções grandes e genéricas. Isso ocorre porque funções menores têm menos linhas de código para serem executadas e podem ser otimizadas de forma mais eficiente pelo mecanismo JavaScript. Além disso, ao separar funções, você pode chamar apenas as funções necessárias em determinadas situações, evitando a execução de código desnecessário.

Então nesse momento iremos criar funções para separar as funcionalidades do nosso código e iniciar o tratamento de erros possíveis do nosso projeto.

O tratamento de erros em JavaScript é uma técnica utilizada para lidar com erros que podem ocorrer durante a execução de um programa. Esses erros podem ser causados por diversos motivos, como erros de sintaxe, erros de lógica ou erros de tempo de execução.

THROW NEW ERROR()

Quando você utiliza a expressão `throw new Error()` em JavaScript, você está lançando uma exceção personalizada. Uma exceção é um objeto que contém informações sobre um erro que ocorreu durante a execução do programa. Ao lançar uma exceção com `throw new Error()`, você está sinalizando que algo inesperado aconteceu e que o fluxo normal do programa não pode continuar.

A expressão `new Error()` cria um novo objeto de erro. Você pode passar uma mensagem como argumento para descrever o erro que ocorreu. Essa mensagem será armazenada na propriedade `message` do objeto de erro.

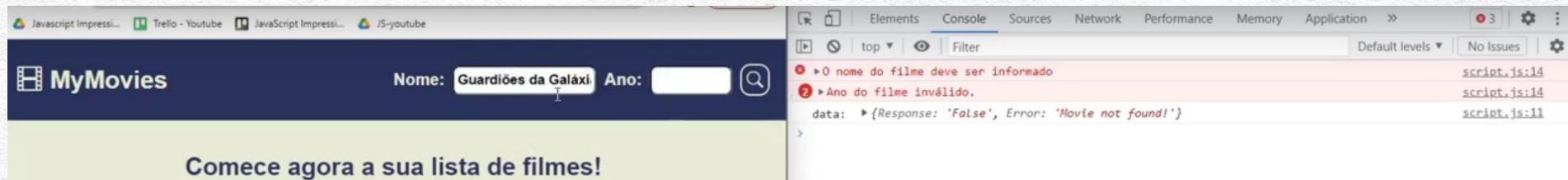
Fazendo os tratamentos de erros com os elementos de busca de Título do Filme e Ano, nosso código terá essa estrutura:

```
6  async function searchButtonClickHandler() {
7    try {
8      let url = `http://www.omdbapi.com/?apikey=${key}&t=${movieNameParameterGenerator()}${movieYearParameterGenerator()}`;
9      const response = await fetch(url);
10     const data = await response.json();
11     console.log('data: ', data);
12     overlay.classList.add('open');
13   } catch (error) {
14     console.error(error.message)
15   }
16 }
17
18 function movieNameParameterGenerator() {
19   if (movieName.value === '') {
20     throw new Error('O nome do filme deve ser informado');
21   }
22   return movieName.value.split(' ').join('+');
23 }
24
25 function movieYearParameterGenerator() {
26   if (movieYear.value === '') {
27     return '';
28   }
29   if (movieYear.value.length !== 4 || Number.isNaN(Number(movieYear.value))) {
30     throw new Error('Ano do filme inválido.');
31   }
32   return `&y=${movieYear.value}`;
33 }
34
35 searchButton.addEventListener('click', searchButtonClickHandler);
```



Módulo 14 – Notificando o usuário de erros (1 / 4)

Anteriormente começamos com o tratamento de erros do nosso projeto, porém a API retorna alguns erros também que precisam ser tratados separadamente, e para fazer esse tratamento vamos utilizar a biblioteca NOTIE:



Vamos importar o NOTIE para o arquivo HTML:

```

OPEN EDITORS 1 unsaved
# style.css
index.html
JS script.js M
JS modal.js
# modal.css
● {} Untitled-1
QUERO... .vscode
> .vscode
src\notie-master
dist
# notie.css
JS notie.js
# notie.min.css
JS notie.min.js
> src
.B .babelrc
.eslintignore
.eslintrc
.gitignore
demo.gif
gulpfile.js
LICENSE.md
{} package-lock.json
{} package.json
README.md

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta http-equiv="X-UA-Compatible" content="IE=edge" />
6     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
7     <title>Lista de filmes para assistir</title>
8     <link rel="stylesheet" href="./style.css" />
9     <link
10        rel="stylesheet"
11        href="https://cdn.jsdelivr.net/npm/bootstrap-icons@1.10.5/font/bootstrap-icons.css"
12      />
13     <link rel="stylesheet" href="./modal.css" />
14     <link rel="stylesheet" href="../src/notie-master/dist/notie.min.css" />
15   </head>
64   |   </article>
65   |   </main>
66   |   <script src="./script.js"></script>
67   |   <script src="./modal.js"></script>
68   |   <script src="./key.js"></script>
69   |   <script src="../src/notie-master/dist/notie.js"></script>
70   |   </body>
71   </html>

```

Módulo 14 – Notificando o usuário de erros (2 / 4)

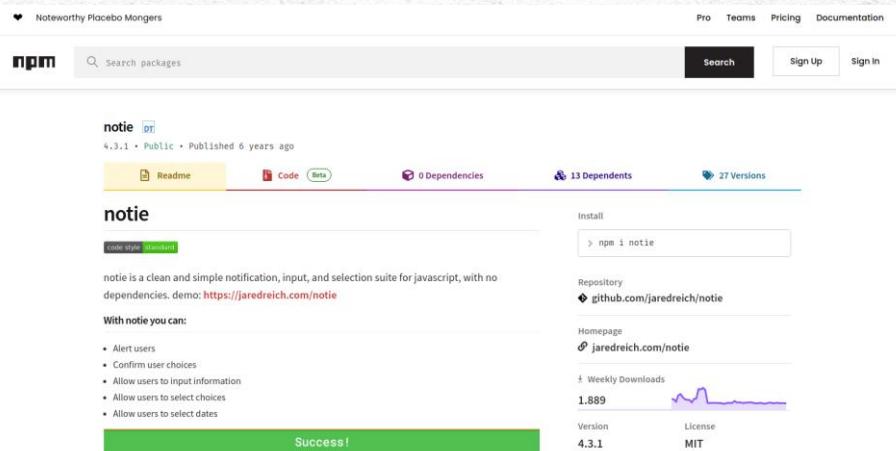
NPM

O **npm (Node Package Manager)** é um gerenciador de pacotes para a plataforma Node.js. Ele é uma ferramenta que permite instalar, gerenciar e compartilhar bibliotecas e módulos de código JavaScript desenvolvidos por terceiros.

Com o npm, você pode facilmente adicionar dependências aos seus projetos Node.js. Essas dependências são pacotes de código JavaScript que fornecem funcionalidades específicas, como frameworks, bibliotecas de utilidades, plugins e muito mais. O npm permite que você especifique as dependências necessárias em um arquivo chamado package.json, que é um arquivo de configuração do seu projeto.

Além de instalar pacotes, o npm também fornece recursos para atualizar, remover e gerenciar as versões das dependências. Ele possui um vasto repositório online chamado npm Registry, onde você pode encontrar milhares de pacotes prontos para uso.

O npm é amplamente utilizado na comunidade JavaScript e é uma parte essencial do ecossistema do Node.js. Ele simplifica o processo de gerenciamento de dependências e facilita a colaboração e compartilhamento de código entre desenvolvedores.



Método notie.alert()

O **método notie.alert()** é usado para exibir uma notificação de alerta. Ele recebe um objeto como argumento com as seguintes propriedades:

- **type** (obrigatório): Especifica o tipo de alerta. Os valores possíveis são "success" (sucesso), "warning" (aviso), "error" (erro), "info" (informação) e "neutral" (neutro).
- **text** (obrigatório): Especifica o conteúdo de texto do alerta.
- **time** (opcional): Especifica a duração em segundos para o alerta ser automaticamente fechado. Se não for fornecido, o alerta não será fechado automaticamente.
- **position** (opcional): Especifica a posição do alerta. Os valores possíveis são "top" (topo), "bottom" (fundo), "center" (centro), "middle" (meio), "left" (esquerda), "right" (direita) ou uma combinação desses valores. A posição padrão é "top" (topo).

Você pode personalizar as cores, tamanhos de fonte e outros estilos do alerta modificando as variáveis no início do arquivo notie.js.

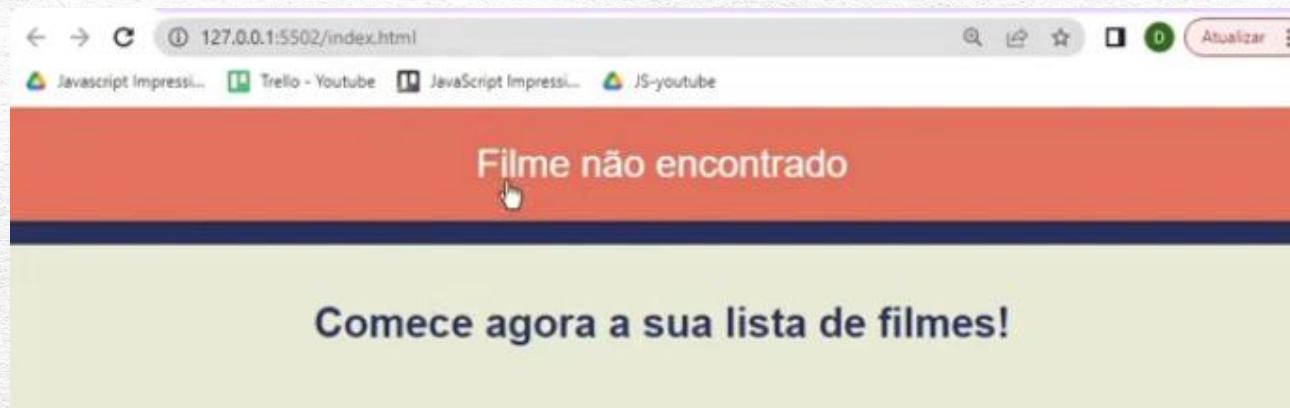
Available methods:

```
notie.alert({
  type: Number|String, // optional, default = 4, enum: [1, 2, 3, 4, 5, 'su
  text: String,
  stay: Boolean, // optional, default = false
  time: Number, // optional, default = 3, minimum = 1,
  position: String // optional, default = 'top', enum: ['top', 'bottom']
})
```



Para adicionar a funcionalidade do método `notie.alert` iremos aplicar o seguinte código e verificar o seu retorno:

```
try {
  let url = `http://www.omdbapi.com/?apikey=${key}&t=${movieNameParameterGenerator()}${movieYearParameterGenerator()}`;
  const response = await fetch(url);
  const data = await response.json();
  console.log('data: ', data);
  if (data.Error) {
    throw new Error('Filme não encontrado');
  }
  overlay.classList.add('open');
} catch (error) {
  notie.alert({ type: 'error', text: error.message });
}
```



Para deixar nosso Modal dinâmico e obtendo o retorno diretamente da API recortamos os elementos HTML que fazem parte dele para adicionar funcionalidade e deixar nosso código mais legível e de fácil manutenção.

E colocaremos essa funcionalidade dentro do arquivo modal.js.

Propriedade innerHTML

A **propriedade innerHTML** em JavaScript é usada para obter ou definir o conteúdo HTML de um elemento. Ela permite manipular o conteúdo interno de um elemento HTML, incluindo texto, tags HTML e outros elementos filhos. Quando você usa **elemento.innerHTML**, você está acessando o conteúdo HTML dentro do elemento especificado.

```
8  function createModal() {
9    modalContainer.innerHTML =
10   <h2 id="movie-title">The Godfather - 1972</h2>
11   <section id="modal-body">
12     
17   <div id="movie-info">
18     <h3 id="movie-plot">
19       The aging patriarch of an organized crime dynasty in postwar New
20       York City transfers control of his clandestine empire to his
21       reluctant youngest son.
22     </h3>
23   <div id="movie-cast">
24     <h4>Elenco:</h4>
25     <h5>Marlon Brando, Al Pacino, James Caan</h5>
26   </div>
27   <div id="movie-genre">
28     <h4>Gênero:</h4>
29     <h5>Crime, Drama</h5>
30   </div>
31   </div>
32 </section>
33 <section id="modal-footer">
34   <button id="add-to-list">Adicionar à Lista</button>
35 </section>;      You, 1 second ago • Uncommitted changes
36 }
```

Agora vamos utilizar os dados que recebemos da resposta da nossa API, lembrando que recebemos um objeto e através das chaves conseguimos pegar seu valores utilizando String Template.

```
data: script.js:11
  ▼ {Title: 'Godfather', Year: '2022', Rated: 'TV-14', Released: '05 Oct 2022', Runtime: '157 min', ...} ⓘ
    Actors: "Chiranjeevi, Salman Khan, Nayanthara"
    Awards: "1 win & 1 nomination"
    BoxOffice: "N/A"
    Country: "India"
    DVD: "N/A"
    Director: "Mohan Raja"
    Genre: "Action, Crime, Drama"
    Language: "Tamil, Kannada, Malayalam, Telugu, Hindi"
    Metascore: "N/A"
    Plot: "After the death of a political leader, a mysterious man steps in to ascend the throne."
    Poster: "https://m.media-amazon.com/images/M/MV5BYmIxMjM2N2MtYjJiMC00NDNmLWExMDEtZjYyYjIyNjMzMDEwXkEyXkFqc
    Production: "N/A"
    Rated: "TV-14"
    ▶ Ratings: [...]
    Released: "05 Oct 2022"
    Response: "True"
    Runtime: "157 min"
    Title: "Godfather"
    Type: "movie"
    Website: "N/A"
    Writer: "Lakshmi Bhupala, Murali Gopy, Mohan Raja"
    Year: "2022"
    imdbID: "tt13130308"
    imdbRating: "5.2"
    imdbVotes: "7,093"
    ▶ [[Prototype]]: Object
```



As informações que queremos buscar para montar nosso Modal seguirá esse modelo de código, e também iremos aplicar a funcionalidade no arquivo script.js na função que irá lidar com o botão do elemento de busca e API:

```
8  function createModal(data) {
9    modalContainer.innerHTML =
10   <h2 id="movie-title">${data.Title} - ${data.Year}</h2>
11   <section id="modal-body">
12     <img
13       id="movie-poster"
14       src=${data.Poster}
15       alt="Poster do Filme."
16     />
17     <div id="movie-info">
18       <h3 id="movie-plot">
19         ${data.Plot}
20       </h3>
21       <div id="movie-cast">
22         <h4>Elenco:</h4>
23         <h5>${data.Actors}</h5>
24       </div>
25       <div id="movie-genre">
26         <h4>Gênero:</h4>
27         <h5>${data.Genre}</h5>
28       </div>
29     </div>
30   </section>
31   <section id="modal-footer">
32     <button id="add-to-list">Adicionar à Lista</button>
33   </section>`;
34 }
```

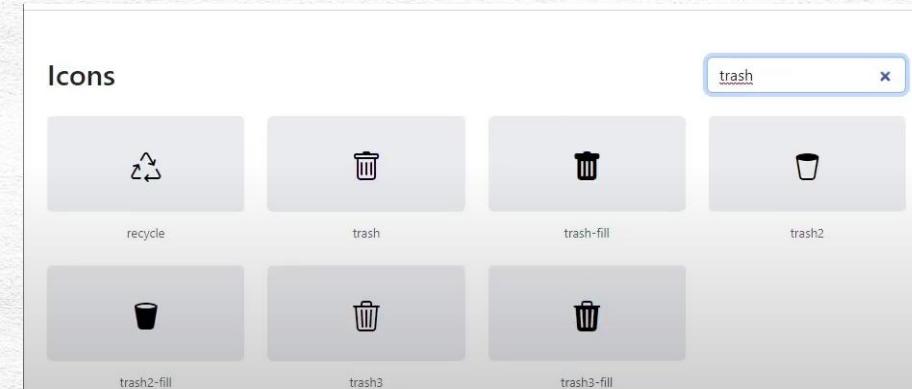
```
JS script.js > ⚡ searchButtonClickHandler
You, 1 second ago | 1 author (You)
1 const searchButton = document.getElementById('search-button');
2 const overlay = document.getElementById('modal-overlay');
3 const movieName = document.getElementById('movie-name');
4 const movieYear = document.getElementById('movie-year');
5
6 async function searchButtonClickHandler() {
7   try {
8     let url = `http://www.omdbapi.com/?apikey=${key}&t=${movieNameParameterGenerator()}${movieYearParameterGenerator()}`;
9     const response = await fetch(url);
10    const data = await response.json();
11    console.log('data: ', data);
12    if (data.Error) {
13      throw new Error('Filme não encontrado');
14    }
15    createModal(data);
16    overlay.classList.add('open');
17  } catch (error) {
18    notie.alert({ type: 'error', text: error.message });
19  }
20 }
```



Preparando o nosso projeto para adicionar funcionalidade ao botão de "Adicionar filme á lista" aplicaremos novamente o conceito de MOCK, ou seja, vamos iniciar com um Modal / filme fixo para fazer a estrutura de como queremos que nosso programa se comporta e depois adicionaremos a inteligência para adicionar os filmes da requisição que fizermos através da busca.

Vamos adicionar os elementos novos a nossa página principal e novamente entrar no site do Bootstrap para pegar o ícone de lixeira para depois implementarmos a lógica no botão de remover o filme da lista:

```
17 <header>
18   <h2><i class="bi bi-film"></i>MyMovies</h2>
19   <section id="search-container">
20     <div class="input-wrapper movie-name-wrapper">
21       <label for="movie-name">Nome:</label>
22       <input type="text" id="movie-name" />
23     </div>
24     <div class="input-wrapper movie-year-wrapper">
25       <label for="movie-year">Ano:</label>
26       <input type="text" id="movie-year" />
27     </div>
28     <button id="search-button"><i class="bi bi-search"></i> Buscar</button>
29   </section>
30 </header>
31 <main>
32   <h2>Comece agora a sua lista de filmes!</h2>
33   <section id="movie-list">
34     <article>
35       
39       <button>Remover</button> You, 1 second ago
40     </article>
41   </section>
```



Módulo 14 – Preparando a lista de filmes (2 / 2)

Propriedade gap

A **propriedade gap** em CSS é usada para definir o espaçamento entre os itens de um grid ou flex container. Ela permite controlar o espaço entre as linhas e colunas de um layout.

A **propriedade gap** aceita um ou dois valores, que representam o espaçamento horizontal e vertical, respectivamente. Se apenas um valor for fornecido, ele será aplicado tanto para o espaçamento horizontal quanto para o vertical. Os valores podem ser especificados em unidades de medida, como pixels (px), porcentagem (%) ou em (unidades relativas ao tamanho da fonte).

A **propriedade gap** é especialmente útil quando se trabalha com layouts de grade (grid) ou flexíveis (flex), onde você deseja adicionar espaçamento entre os itens. Por exemplo, se você tiver um contêiner de grade com várias células, pode usar gap para definir o espaçamento entre elas.

Ajustaremos esteticamente nossos elementos no arquivo style.css:

```
--  
100 #movie-list {  
101   display: flex;  
102   flex-wrap: wrap;  
103   justify-content: center;  
104   gap: 2rem;  
105   margin-top: 1rem;  
106 }  
107  
108 #movie-list article {  
109   display: flex;  
110   flex-direction: column;  
111 }  
112  
113 #movie-list article > img {  
114   border-radius: 10px;  
115   margin-bottom: 0.5rem;  
116 }  
117  
118 .remove-button {  
119   border: 3px solid var(--secondary-color);  
120   padding: 0.3rem 0;  
121   width: 100%;  
122   border-radius: 10px;  
123   font-weight: bold;  
124   box-shadow: 1px 1px 10px #000;  
125   color: var(--secondary-color);  
126 }  
127  
128 .remove-button:focus,  
129 .remove-button:hover {  
130   background-color: var(--light-secondary);  
131   color: var(--lighest-color);  
132 }  
133 }
```



Para adicionar a inteligência do botão de adicionar do nosso Modal, vamos criar a lógica no arquivo script.js

```
js script.js > ...
1 const searchButton = document.getElementById('search-button');
2 const overlay = document.getElementById('modal-overlay');
3 const movieName = document.getElementById('movie-name');
4 const movieYear = document.getElementById('movie-year');
5 const movieListContainer = document.getElementById('movie-list');
6
7 let movieList = [];
8
```

Push()

O **método push()** em JavaScript é usado para adicionar um ou mais elementos ao final de um array. Ele modifica o array original, aumentando seu tamanho e adicionando os novos elementos no final.

E criar uma nova função que irá armazenar a lógica para adicionarmos um filme ao clicarmos no botão.

```
42 function addToList(data) {
43   movieList.push(data);
44 }
45
```



Atributo onclick

O **atributo onclick** em HTML é usado para associar um evento de clique a um elemento HTML, como um botão, link ou qualquer outro elemento interativo. Ele permite que você especifique o código JavaScript que será executado quando o elemento for clicado.

O **atributo onclick** pode ser usado em vários elementos HTML, como botões, links, imagens e até mesmo em elementos de formulário, como caixas de seleção e campos de entrada.

É importante lembrar que o código JavaScript especificado no atributo onclick será executado quando o evento de clique ocorrer. Portanto, você pode usar esse atributo para realizar ações personalizadas, manipular dados, exibir mensagens ou realizar qualquer outra tarefa desejada quando um elemento for clicado.

No entanto, é recomendado separar o código JavaScript do HTML, seguindo as melhores práticas de separação de preocupações. Em vez de usar o atributo onclick, é preferível adicionar um ouvinte de evento usando JavaScript para manter o código mais organizado e modular.

Método JSON.stringify()

O **método JSON.stringify()** em JavaScript é usado para converter um objeto JavaScript em uma string JSON. Ele serializa o objeto, transformando suas propriedades em uma representação de string no formato JSON.

A sintaxe básica do **método JSON.stringify()** é a seguinte: JSON.stringify(objeto, propriedades, espaçamento);

Onde:

- **objeto** é o objeto JavaScript que você deseja converter em uma string JSON.
- **propriedades** (opcional) é um array ou uma função que especifica as propriedades a serem incluídas na string JSON resultante. Se for omitido, todas as propriedades enumeráveis do objeto serão incluídas.
- **espaçamento** (opcional) é usado para formatar a string JSON com espaçamento e recuo. Pode ser um número para especificar o número de espaços de recuo ou uma string para usar como recuo. Se for omitido, a string JSON será compactada sem espaçamento adicional.

O **método JSON.stringify()** é útil quando você precisa enviar dados para um servidor ou armazená-los em um formato de string JSON. Ele permite que você converta objetos JavaScript em uma representação JSON para facilitar a comunicação e o armazenamento de dados estruturados.

Método replace()

O **método replace()** em JavaScript é usado para substituir parte de uma string por outra string. Ele procura por um padrão especificado na string original e substitui todas as ocorrências desse padrão pela nova string fornecida.

A sintaxe básica do método replace() é a seguinte: `string.replace(padrao, novaString);`

Onde:

- **string** é a string original na qual você deseja fazer a substituição.
- **padrao** é o padrão que você deseja substituir na string original. Pode ser uma string ou uma expressão regular.
- **novaString** é a string que será usada para substituir o padrão encontrado na string original.

O **método replace()** também pode usar grupos de captura e funções de substituição mais avançadas para realizar substituições mais complexas. Consulte a documentação do JavaScript para obter mais informações sobre esses recursos.

O **método replace()** é útil quando você precisa substituir partes específicas de uma string por outra string. Ele permite que você faça substituições de forma fácil e flexível, tornando a manipulação de strings mais eficiente.

Aplicando esses conceitos que acabamos de aprender na lógica do nosso botão de adicionar um filme a lista:

```
31     <section id="modal-footer">
32         <button id="add-to-list" onclick='{addToList(${JSON.stringify(
33             |   data
34             ).replace("'", ``)})}'>Adicionar à Lista</button>
35     </section>;
36 }
```

Para finalizarmos a funcionalidade do botão de adicionar o filme, aplicaremos a inteligência para retornar os filmes de forma dinâmica.

```
41
42  function addToList(data) {
43    if (isFilmAlreadyOnTheList(data.imdbID)) {
44      notie.alert({ type: 'error', text: 'Filme já está na lista' });
45      return;
46    }
47    movieList.push(data);
48    updateUI(data);
49    overlay.classList.remove('open');
50  }
51
52  function updateUI(data) {
53    movieListContainer.innerHTML += `<article id='movie-card-${data.imdbID}'>
54      
58      <button class="remove-button" onclick='{removeFilmFromList("${data.imdbID}")}'>
59        <i class="bi bi-trash"></i> Remover
60      </button>
61    </article>`;
62  }
63
```

Módulo 14 – Identificando os filmes de forma única (1 / 2)

Repare que no nosso código utilizamos uma chave na nossa variável data de .imdbID, essa chave representa o ID do nosso filme, ou seja, um ID (identificador) é um valor único usado para identificar algo de forma exclusiva. É como um nome único atribuído a algo para que possa ser facilmente identificado e referenciado.

Então através do ID conseguimos criar uma lógica para que não adicionemos filmes que já estejam na nossa lista.

```
function addToList(data) {
  isFilmAlreadyOnTheList([data.imdbID])
  movieList.push(data);
  updateUI(data);
  overlay.classList.remove('open');
}
```

Função que irá adicionar verificando o ID do filme.

Função que traz os elementos HMTL de forma dinâmica

```
function updateUI(data) {
  movieListContainer.innerHTML += `<article id='movie-card-${data.imdbID}'>
    
    <button class="remove-button">
      <i class="bi bi-trash"></i> Remover
    </button>
  </article>`;
}
```

```
function isFilmAlreadyOnTheList(imdbId) {
  function isThisIdFromThisMovie(movie) {
    return movie.imdbID === imdbId;
  }
  return movieList.find(isThisIdFromThisMovie);
}
```

Função que verifica o ID e será utilizada na addToList.



Para finalizarmos a adição dos filmes a nossa lista, iremos aplicar a biblioteca Notie para que nos alerte caso o filme já tenha sido adicionado.

```
41
42  function addToList(data) {
43    if (isFilmAlreadyOnTheList(data.imdbID)) {
44      notie.alert({ type: 'error', text: 'Filme já está na lista' });
45      return;
46    }
47    movieList.push(data);
48    updateUI(data);
49    overlay.classList.remove('open');
50  }
```

Agora vamos adicionar a lógica de remover o filme da lista.

Método remove()

O **método remove()** em JavaScript é usado para remover um elemento HTML do documento. Ele faz parte da interface `ChildNode` e pode ser chamado em qualquer elemento HTML que seja um nó filho de outro elemento.

Ele remove o elemento e todos os seus descendentes (elementos filhos e seus filhos, e assim por diante).

É importante observar que o método `remove()` não é suportado em versões mais antigas do Internet Explorer. Se você precisa oferecer suporte a navegadores mais antigos, pode usar uma abordagem alternativa, como remover o elemento pai usando o método `removeChild()` do elemento pai.

```
function removeFilmFromList(imdbId) {
  movieList = movieList.filter((movie) => movie.imdbID !== imdbId);
  document.getElementById(`movie-card-${imdbId}`).remove();
}
```



```
function removeFilmFromList(imdbId) {  
    movieList = movieList.filter((movie) => movie.imdbID !== imdbId);  
    document.getElementById(`movie-card-${imdbId}`).remove();  
}
```

A expressão **.filter(movie => movie.imdbId !== imdb)** é uma função de filtro em JavaScript que é aplicada a um array de objetos. Essa função é usada para filtrar os elementos do array com base em uma condição específica.

Vamos analisar a expressão em partes:

- **.filter():** é um método de array em JavaScript que cria um novo array com todos os elementos que passam por um teste (função de retorno de chamada) fornecido. Ele retorna um novo array contendo apenas os elementos que atendem à condição especificada.
- **movie => movie.imdbId !== imdb:** é uma função de retorno de chamada (callback) que é aplicada a cada elemento do array. Nesse caso, a função verifica se o valor da propriedade `imdbId` do objeto `movie` é diferente do valor da variável `imdb`. Se a condição for verdadeira, o elemento é mantido no novo array resultante. Caso contrário, o elemento é filtrado e não é incluído no novo array.

Portanto, a expressão **.filter(movie => movie.imdbId !== imdb)** é usada para filtrar um array de objetos, mantendo apenas os objetos cuja propriedade `imdbId` é diferente do valor da variável `imdb`. O resultado é um novo array contendo apenas os objetos que não correspondem ao valor especificado.

Finalizando nosso botão de remover vamos aplicar a inteligência ao elemento do botão do HTML.

```
function updateUI(data) {
  movieListContainer.innerHTML += `<article id='movie-card-${data.imdbID}'>
    
    <button class="remove-button" onclick='{removeFilmFromList("${data.imdbID}")}'>
      <i class="bi bi-trash"></i> Remover
    </button>
  </article>`;
}
```

Para finalizarmos nosso projeto inicialmente vamos fazer alguns ajustes finais de CSS, primeiro no arquivo style.css:

Alinhar os modais dos filmes da nossa lista:

```
107  
108  #movie-list article {  
109    display: flex;  
110    flex-direction: column;  
111    justify-content: space-between;  
112 }
```



Aplicar um tamanho fixo ao elemento de ícone da busca:



```
79  
80  #search-button {  
81    width: 20px;      You, 1 second ago - Un  
82    padding: 0.3rem;  
83    border: 2px solid var(--lighest-color);  
84    border-radius: 10px;  
85 }
```



Módulo 14 – Ajustes finais / armazenando a lista no navegador (2 / 8)

E ajustar o nosso arquivo modal.css para melhor a visualização do Modal do filme:



```
61 #movie-info {  
62   display: flex;  
63   flex-direction: column;  
64   justify-content: space-around;  
65   max-width: 300px;  
66   padding-left: 2rem; You can add padding-left here  
67 }  
68
```



Podemos perceber que quando a página é atualizada perdemos as informações dos filmes adicionados a nossa lista para corrigir e conseguirmos armazenar a nossa lista vamos aprender sobre LocalStorage.

LocalStorage

O **LocalStorage** é uma funcionalidade do JavaScript que permite armazenar dados no navegador da web. Ele fornece uma maneira simples de armazenar dados no formato de pares chave-valor, semelhante a um dicionário.

O **LocalStorage** é uma forma de armazenamento persistente no navegador, o que significa que os dados armazenados permanecem disponíveis mesmo após o navegador ser fechado e reaberto. Isso o torna útil para armazenar informações como preferências do usuário, configurações, dados de login e outras informações que precisam ser mantidas entre sessões.

Para usar o **LocalStorage**, você pode usar os métodos **setItem**, **getItem** e **removeItem** do objeto localStorage.

É importante observar que o **LocalStorage** armazena apenas valores como strings. Portanto, se você precisar armazenar outros tipos de dados, como objetos JavaScript, será necessário convertê-los em strings usando JSON.stringify antes de armazená-los e, em seguida, usar JSON.parse para convertê-los de volta para o formato original ao recuperá-los.

Além disso, o **LocalStorage** tem um limite de armazenamento, geralmente em torno de 5MB, que varia de navegador para navegador. Portanto, é importante ter isso em mente ao decidir o que armazenar no LocalStorage.

Método `setItem`

O método `setItem` do `LocalStorage` é usado para armazenar um par chave-valor no armazenamento local do navegador. Ele recebe dois argumentos: a chave e o valor que você deseja armazenar.

Ao usar o `setItem`, se você já tiver um valor armazenado com a mesma chave, ele será substituído pelo novo valor. Portanto, tenha cuidado ao escolher as chaves para evitar conflitos.

Método `getItem`

O método `getItem` do `LocalStorage` é usado para recuperar o valor armazenado em uma determinada chave no armazenamento local do navegador. Ele recebe um argumento, que é a chave do valor que você deseja recuperar.

Método `removeItem`

O método `removeItem` do `LocalStorage` é usado para remover um item armazenado no armazenamento local do navegador. Ele recebe um argumento, que é a chave do item que você deseja remover.

JSON.parse

O **método JSON.parse** é uma função do JavaScript que converte uma string JSON em um objeto JavaScript. Ele é usado quando você precisa transformar dados em formato JSON em um formato que possa ser manipulado e utilizado em seu código.

É importante observar que o JSON deve estar em um formato válido para que o **JSON.parse** funcione corretamente. Caso contrário, uma exceção será lançada. Certifique-se de que a string JSON esteja bem formada, com aspas duplas em torno das chaves e valores, e que siga a sintaxe correta do JSON.

Além disso, o **JSON.parse** também pode ser usado para converter outros tipos de dados JSON, como arrays JSON, em objetos JavaScript. Ele é uma ferramenta poderosa para trabalhar com dados em formato JSON em seu código.

Operador de coalescência nula

O **operador de coalescência nula (??)** é usado para fornecer um valor padrão quando um valor é nulo ou indefinido. Ele retorna o primeiro operando se não for nulo ou indefinido, caso contrário, retorna o segundo operando.

Arquivo script.js

```
7 | let movieList = JSON.parse(localStorage.getItem('movieList')) ?? [];
```



Vamos criar a função que será responsável pela funcionalidade de armazenar os filmes no LocalStorage e depois aplica-la quando adicionamos ou removemos o filme da lista, ou seja, essa atualização ficará na lógica dos nossos botões.

```
76 | function updateLocalStorage() {
77 |   localStorage.setItem('movieList', JSON.stringify(movieList));
78 | }
79 |
```

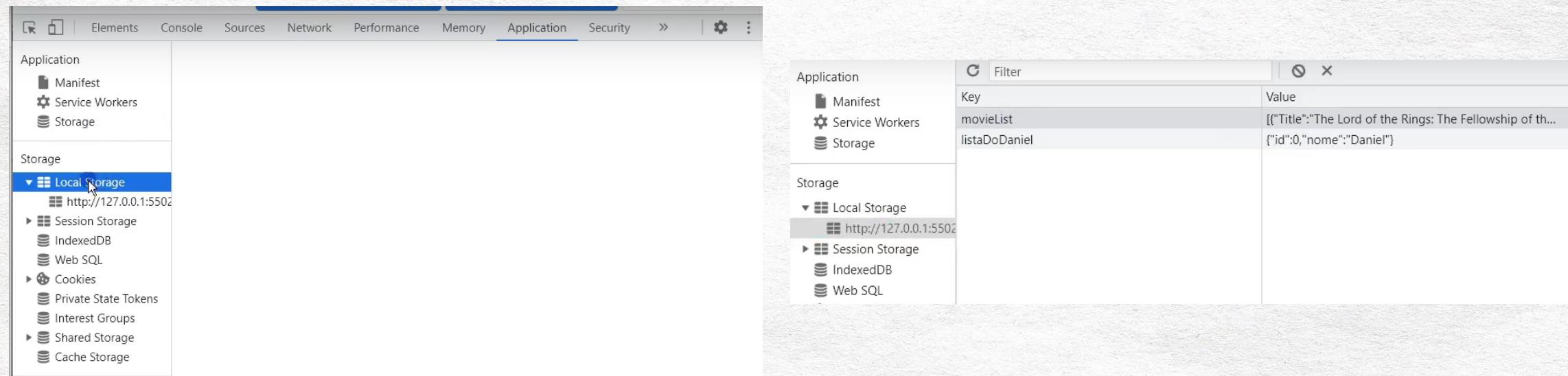
Adicionando um filme

```
42 | function addToList(data) {
43 |   if (isFilmAlreadyOnTheList(data.imdbID)) {
44 |     notie.alert({ type: 'error', text: 'Filme já está na lista' });
45 |     return;
46 |   }
47 |   movieList.push(data);
48 |   updateLocalStorage(); You, 1 second ago • Uncommitted changes
49 |   updateUI(data);
50 |   overlay.classList.remove('open');
51 | }
```

Removendo um filme

```
72 | function removeFilmFromList(imdbId) {
73 |   movieList = movieList.filter((movie) => movie.imdbID !== imdbId);
74 |   document.getElementById(`movie-card-${imdbId}`).remove();
75 |   updateLocalStorage(); You, 1 second ago • Uncommitted changes
76 | }
```

Para você visualizar como se o **LocalStorage** está funcionando e como ele armazena, vamos utilizar o DevTools (F12), na aba de **Application**



Método **forEach**

O **forEach** é um método disponível em arrays na linguagem JavaScript que permite percorrer cada elemento do array e executar uma função para cada um deles. Ele é uma forma mais concisa e conveniente de iterar sobre os elementos de um array em comparação com o uso de um loop for tradicional.

A principal vantagem do **forEach** é que ele abstrai a lógica de iteração e permite que você se concentre na manipulação dos elementos individuais do array. Além disso, o forEach também é uma forma mais legível e expressiva de percorrer um array em comparação com um loop for tradicional.

É importante mencionar que o forEach não retorna um novo array.

Para finalizar nossa implementação:

```
81  
82 movieList.forEach(updateUI);  
83
```

Pronto! Parabéns finalizamos o nosso primeiro projeto!

Caso você queira acompanhar o código de forma global você pode fazer o download dos arquivos que estão na plataforma do curso.