# JAVASCRIPT IMPRESSIONADOR

## Curso Completo de JavaScript da Hashtag Programação

# SUMÁRIO

## MÓDULO 2 DADOS, VARIÁVEIS E OPERAÇÕES


01 PRIMEIRO PROGRAMA ..... 008
02 VARIÁVEIS E CONSTANTES ..... 018
03 DECLARANDO E DEFININDO VARIÁVEIS ..... 025
04 OPERADORES ARITMÉTICOS EM JAVASCRIPT ..... 032
05 PADRONIZANDO O CÓDIGO ..... 040
06 TIPOS DE DADOS: STRINGS E NUMBERS ..... 048
07 NÚMEROS BINÁRIOS E BITS ..... 053
08 BOOLEANS: VARIÁVEIS E OPERAÇÕES ..... 057
09 OPERADORES DE COMPARAÇÃO EM JAVASCRIPT ..... 061
10 UNDEFINED E NULL ..... 066
11 OBJETCT EM JAVASCRIPT ..... 070
12 ARRAY EM JAVASCRIPT ..... 075
13 NOMENCLATURA E MUTABILIDADE ..... 079
14 POR QUE "TIPOS REFERÊNCIAS" ..... 083
15 AS CONSEQUÊNCIAS DA MANIPULAÇÃO DE REFERÊNCIAS ..... 085


## MÓDULO 3 FUNÇÕES


01 A FUNÇÃO ..... 092
02 PARÂMETROS E ARGUMENTOS ..... 095
03 RETORNO ( RETURN) ..... 101
04 MÚLTIPLOS PARÂMETROS E ORDEM DE EXECUÇÃO ..... 106
05 ESCOPOS ..... 111
06 O QUE SÃO MÉTODOS? ..... 122
07 MÉTODOS ASSOCIADOS A TIPOS ..... 129
08 MÉTODOS DE ARRAYS ..... 137


## MÓDULO 4 DEPURAÇÃO E DESENVOLVIMENTO EFICIENTE


01 OS VÁRIOS TIPOS DE ERROS ..... 141
02 CONSOLE LOG: UMA PRIMEIRA ABORDAGEM ..... 149
03 DENTRO DO NAVEGADOR ..... 159

# SUMÁRIO

## MÓDULO 4 DEPURAÇÃO E DESENVOLVIMENTO EFICIENTE


04 DEBUGANDO SEU CÓDIGO ..... 178

05 DEV TOOLS NO VS CODE ..... 188


## MÓDULO 5 ESTRUTURAS CONDICIONAIS E DE REPETIÇÃO


01 INTRODUÇÃO / CONTROLE DE FLUXO DE EXECUÇÃO ..... 204

02 BLOCOS CONDICIONAIS - IF ..... 207

03 IF / ELSE IF / ELSE ..... 212

04 SWITCH CASE ..... 219

05 OPERADOR TERNÁRIO ..... 226

06 ESTRUTURAS DE REPETIÇÃO ..... 232

07 DO... WHILE ..... 233

08 WHILE ..... 237

09 FOR ..... 247

10 RECURSIVIDADE ..... 252

11 O PROBLEMA DA RECURSIVIDADE ..... 260

12 ENUMERANDO INFORMAÇÕES EM OBJETOS ..... 262

13 ENUMERANDO INFORMAÇÕES EM ARRAYS ..... 265

14 VALORES TRUTHY E FALSY ..... 268


## MÓDULO 6 DESENVOLVIMENTO WEB


01 COMO A COMUNICAÇÃO DA INTERNET FUNCIONA ..... 271

02 ARQUITETURAS E PROTOCOLOS DA INTERNET ..... 276

03 O HTML ..... 283

04 TAGS E SUAS INFORMAÇÕES ..... 292

05 O CSS ..... 296

06 PROPRIEDADES EM CSS ..... 300

07 PROBLEMAS DESSA PRIMEIRA ABORDAGEM ..... 303

08 SEPARAÇÃO DE RESPONSABILIDADES > CRIANDO UM ARQUIVO CSS 306

# SUMÁRIO

## MÓDULO 6 DESENVOLVIMENTO WEB


09 INLINE STYLING, SELETORES, CLASSES E IDENTIFICADORES ..... 308
10 SELETORES CSS ..... 311
11 O "BOX MODEL" ..... 316
12 PRINCIPAIS PARÂMETROS ESTILIZÁVEIS ..... 322


## MÓDULO 7 PRIMEIRO PROJETO


01 APRESENTAÇÃO DO PROJETO ..... 329
02 RECURSOS EXTERNOS UTILIZADOS ..... 332
03 COMEÇANDO A ESTRUTURA DO PROJETO ..... 340
04 CRIANDO O NOSSO HEADER ..... 347
05 ESTILIZANDO O HEADER ..... 354
06 OS ELEMENTOS DE BUSCA ..... 362
07 AJUSTANDO O HEADER PARA DISPLAY MAIORES ..... 367
08 SEÇÃO PRINCIPAL DA PÁGINA ..... 368
09 CRIANDO A ESTRUTURA DO MODAL ..... 373
10 ESTILIZANDO O MODAL ..... 375
11 MOCK DO PRIMEIRO MODAL DE FILME ..... 382
12 CSS DO MODAL PROJETADO ..... 388
13 ADICIONANDO INTELIGÊNCIA AO MODAL ..... 382
14 COMEÇANDO A INTELIGÊNCIA DA PÁGINA PRINCIPAL ..... 398
15 CAPTURANDO AS ENTRADAS DOS USUÁRIOS ..... 401
16 CONSUMINDO A API DO OMDB ..... 405
17 LIDANDO COM ERROS ..... 410
18 NOTIFICANDO O USUÁRIO DE ERROS ..... 413
19 MODAL DINÂMICO ..... 417
20 PREPARANDO A LISTA DE FILMES ..... 420
21 A INTELIGÊNCIA DO BOTÃO DO MODAL ..... 422
22 FINALIZANDO A FUNCIONALIDADE DE ADIÇÃO ..... 427
23 IDENTIFICANDO OS FILMES DE FORMA ÚNICA ..... 428
24 REMOÇÃO DE UM FILME ..... 430
25 AJUSTES FINAIS/ ARMAZENANDO A LISTA NO NAVEGADOR ..... 433

# SUMÁRIO

## MÓDULO 8 TRATAMENTO DE ERRO


01 INTRODUÇÃO Á PROGRAMAÇÃO DEFENSIVA . . . . . 442
02 ANTECIPANDO ERROS COM CONDICIONAIS . . . . . 445
03 TRY, CATCH E THROW . . . . . 450
04 LANÇAR ERRO COMPLETO OU SÓ MENSAGEM? . . . . . 455
05 FINALLY . . . . . 457


## MÓDULO 9 JAVASCRIPT ASSÍCRONO


01 INTRODUÇÃO > PROGRAMAÇÃO SÍNCRONA . . . . . 460
02 PROGRAMAÇÃO ASSÍNCRONA . . . . . 464
03 FUNÇÕES ANÔNIMAS . . . . . 468
04 ARMAZENANDO FUNÇÕES EM VARIÁVEIS . . . . . 471
05 ARROW FUNCITONS . . . . . 472
06 AS DIFERENTES FORMAS DE UMA ARROW FUNCTION . . . . . 474
07 CALLBAKS . . . . . 476
08 PASSAR FUNÇÃO COMO ARGUMENTO VS CHAMAR FUNÇÃO . . . . . 481
09 CALLBACK HELL . . . . . 485
10 PROMISES . . . . . 491
11 PROGRAMAÇÃO ASSÍNCRONA COM PROMISES . . . . . 496
12 TRATANDO ERROS COM PROMISES . . . . . 499
13 ASYNC / AWAIT . . . . . 501
14 MÚLTIPLAS PROMISES . . . . . 504


## MÓDULO 10 OBJETOS


01 CRIANDO UM OBJETO: OBJECT LITERAL . . . . . 509
02 CRIANDO UM OBJETO: FUNÇÃO FÁBRICA . . . . . 512
03 CRIANDO UM OBJETO: CONSTRUTOR . . . . . 515
04 [OPCIONAL] CONHECENTO AS PARTES DO COMPUTADOR . . . . . 518
05 STACK E HEAP . . . . . 520
06 PASSAGEM POR VALOR VS PASSAGEM POR REFERÊNCIA . . . . . 527

# SUMÁRIO

## MÓDULO 10 OBJETOS


07 O OBJETO COMO UM TIPO . . . . . 531

08 O OPERADOR INSTANCEOF . . . . . 536

09 CLASSES EM JAVASCRIPT . . . . . 539

10 CLASSES VS CONSTRUTOR: AS VÁRIAS VERSÕES DO JS . . . . . 542

11 PROGRAMANDO EM DUPLA: APRESENTAÇÃO DO NOSSO PROJETO . . . 546

12 O CONSTRUTOR PADRÃO DO CRONÔMETRO . . . . . 548

13 CRIANDO O CRONÔMETRO . . . . . 551

14 MEMBROS ESTÁTICOS . . . . . 558

15 GETTERS E SETTERS . . . . . 562

16 MEMRBOS PRIVADOS . . . . . 567

17 INDO ALÉM: A REPRESENTAÇÃO VISUAL DO PROJETO . . . . . 568

18 INDO ALÉM: AJUSTES FINAIS . . . . . 574

Módulo 2

# DADOS, VARIÁVEIS E OPERAÇÕES

DADOS, VARIÁVEIS E OPERAÇÕES

DADOS, VARIÁVEIS E OPERAÇÕES

# Módulo 2 - Dados, Variáveis e Operações - Primeiro Programa (1/10)

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.

- **1° Passo:** Criaremos uma pasta chamada JavaScript Impressionador no seu computador (no local que faça sentido para você- exemplo: Pasta Pessoal, Área de Trabalho, Documentos ).
- **2° Passo:** Dentro da pasta JavaScript Impressionador, 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.

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

Ao finalizar o 3° 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.

# Módulo 2 - Dados, Variáveis e Operações - Primeiro Programa (3/10)

- 4° 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.

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.

1

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 **NODE.JS** .

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.

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

## Instrução 1°

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 2°

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 3°

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** ' .

# Instrução 4°

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!'.

```
Get Started    JS primeiroCodigo.js ×
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.

# Módulo 2 - Dados, Variáveis e Operações - Variáveis e Constantes (1/7)

Recapitulando o que vimos anteriormente, aprendemos que o JavaScript possui um comando chamado **console.log()** que tem a função de imprimir na nossa tela as informações passadas entre seus **( )**. E os comandos sempre são executados uma linha de código depois da outra, de cima para baixo.

E para executarmos o nosso código JavaScript precisamos salvá-lo e depois executar dentro do terminal o comando:

**node ./nomeDoArquivo.js**

Perfeito! Mas agora imagine uma situação na qual queremos imprimir diversas vezes a mesma informação, e eu precisasse copiar e colar ou escrever á cada console.og(), concordam que alguns erros poderiam ocorrer, como colar ou digitar algo errado?

Esse é um exemplo simples que de certa forma seria fácil corrigir, mas concorda que ao executarmos esse código teríamos um erro em sua resposta / saída? Agora imagine em uma aplicação muito maior com mais linhas de código, a tarefa teria um processo de identificação de erro muito mais longo.

```
PS C:\Users\danie\JavaScript Impressionador\modulo_2> node ./primeiroCodigo.js
Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro!
Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro!
Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro!
Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em um marinheiro!
Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro!
```

Uma das formas para evitarmos esses possíveis erros é guardando essas informações em caixinhas. Todas as vezes que quisermos utilizar essa informação iremos dizer ao nosso código para utilizar a caixinha que possui a informação.

Essa caixinha é a nossa **VARIÁVEL**, que é onde iremos armazenar valores que serão manipulados por nossos programas JavaScript. E para utilizarmos elas precisamos colocar uma etiqueta nelas , ou seja, dar um **nome simbólico** a nossa variável ( geralmente esse nome terá relação com o que estaremos armazenando dentro dela).

E também precisamos identificar o **tipo de declaração** que aquela variável se caracteriza (**let ou const**).

Então a **estrutura** de uma **variável** será:

**declaração do tipo** + **nome da variável** + **sinal de igual ( = )** + **informação armazenada.**

```
JS primeiroCodigo.js > ...
1   const mensagemTela = 'Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro!';
2
```

E para utilizarmos essa variável dentro do nosso console.log() se tornou muito mais simples:

```
3   console.log(mensagemTela);
4   console.log(mensagemTela);
5   console.log(mensagemTela);
6   console.log(mensagemTela);
7   console.log(mensagemTela);
```

Agora em um cenário que precisasse alterar a informação do meu console.log() seria muito mais simples, ao invés de a cada console.log() eu efetuar a alteração, é só alterar dentro da variável o conteúdo que quero imprimir, já que o comando console.log() irá capturar o valor que está dentro da minha variável e utilizar.

```
JS primeiroCodigo.js > ...
1  const mensagemTela = 'Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro! Eu também fui um fisiculturista amador em meus 40 anos de idade';
2
3  console.log(mensagemTela);
4  console.log(mensagemTela);
5  console.log(mensagemTela);
6  console.log(mensagemTela);
7  console.log(mensagemTela);
```

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    SQL CONSOLE

PS C:\Users\danie\JavaScript Impressionador\modulo_2> node ./primeiroCodigo.js
Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro! Eu também fui um fisiculturista amador em meus 40 anos de idade
Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro! Eu também fui um fisiculturista amador em meus 40 anos de idade
Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro! Eu também fui um fisiculturista amador em meus 40 anos de idade
Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro! Eu também fui um fisiculturista amador em meus 40 anos de idade
Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro! Eu também fui um fisiculturista amador em meus 40 anos de idade
```

Lembre-se sempre de salvar e depois executar o seu programa, para que essas alterações sejam alteradas e salvas dentro do seu programa.

# Módulo 2 - Dados, Variáveis e Operações - Variáveis e Constantes (4/7)

As diferentes formas de declaração de variáveis são utilizadas para o mesmo propósito de armazenar uma informação ou valor, mas é importante entender que existem diferenças entre elas para saber em qual momento devemos utilizar cada tipo de declaração.

## LET

A variável do tipo **let** permite que você declare variáveis que podem receber outros valores. Em outras palavras, o valor original dessa variável pode ser substituído por outra informação, como se a variável fosse atualizada. Sua característica é ser **MUTÁVEL**.

## CONST

A variável do tipo **const** permite que você declare variáveis **constantes.** Em outras palavras, as informações atribuídas a esse tipo de variável não podem ser alteradas. Geralmente utilizamos para guardar informações fixas, que não serão alteradas nunca em nosso código. Sua característica é ser **IMUTÁVEL**.

## VAR

Você pode se deparar com códigos que utilizem o tipo **var** para variáveis ela atua de forma parecida com o tipo let, porém com uma facilidade de alteração muito maior, o que pode ocasionar muitos bugs no seu código, então a partir das atualizações do JavaScript (ES6), ⚠ **o uso do tipo var não é uma boa prática.**

# Módulo 2 - Dados, Variáveis e Operações - Variáveis e Constantes (5/7)

Como podemos observar o exemplo abaixo, nesse momento declaramos nossa variável com o tipo **LET**, pois iremos demonstrar como podemos substituir o valor e verificar como que o JavaScript faz a leitura desses comandos e suas atualizações. Sabemos que para criar uma variável precisamos construir toda sua estrutura iniciando com o tipo de declaração. Mas e para atualizarmos uma variável existente?

```
JS primeiroCodigo.js > ...
1   let mensagemTela = 'Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro! Eu também fui um fisiculturista amador em meus 40 anos de idade';
2
3   console.log(mensagemTela);
4   console.log(mensagemTela);
5   console.log(mensagemTela);
6   console.log(mensagemTela);
7   console.log(mensagemTela);
8
9
10  mensagemTela = 'Agora o que eu quero ser é um idoso filósofo!';
11
12  console.log(mensagemTela);
```

Para atualizarmos os valores de uma variável precisamos apenas chamá-la ou invocá-la pelo seu nome, nesse caso '**mensagemTela**' + **=** + escrevemos o **novo valor** que aquela variável irá armazenar. Lembrando que isso é possível, pois estamos utilizando o tipo **LET**.

O retorno do seu programa seguirá os comandos de cima para baixo, então efetuará a impressão com o valor original 5 vezes , atualizará a variável e irá imprimir o seu valor novo, ou seja, até o momento em que nossa variável não foi atualizada todo o código acima disso será com o seu valor antigo / original.

```
PS C:\Users\danie\JavaScript Impressionador\modulo_2> node ./primeiroCodigo.js
Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro! Eu também fui um fisiculturista amador em meus 40 anos de idade
Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro! Eu também fui um fisiculturista amador em meus 40 anos de idade
Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro! Eu também fui um fisiculturista amador em meus 40 anos de idade
Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro! Eu também fui um fisiculturista amador em meus 40 anos de idade
Olá mundo! Meu nome é Daniel Porto. Eu tenho 87 anos e sempre sonhei em ser um marinheiro! Eu também fui um fisiculturista amador em meus 40 anos de idade
Agora o que eu quero ser é um idoso filósofo!
```

## Atualizar e Salvar

No VS Code toda atualização que não está salva será indicada por uma bolinha branca, você precisa salvar ( **CTRL+S** ) toda alteração que fizer no seu código para depois executar o seu programa com essas novas alterações.

## Copiando o código

Um atalho do VS Code para você copiar a linha de código para cima ou para baixo é utilizar o comando SHIFT + ALT + tecla para baixo ou tecla para cima.

## O ponto e vírgula ( ; )

O ponto e vírgula em JavaScript é opcional, mas é considerado uma boa prática entre os programadores. É utilizado como separador de instruções, apenas nos casos de mais de uma instrução na mesma linha o **;** é obrigatório.

# Módulo 2 - Dados, Variáveis e Operações - Declarando e Definindo Variáveis (1 / 7)

Vamos falar um pouco mais sobre **VARIÁVEIS**, especificamente algumas regras de como devemos escolher as etiquetas delas, ou seja, a sua nomenclatura (nome).
Você irá perceber ao longo do curso que o JavaScript possui boas práticas, ou seja, situações que são opcionais mas que fazem o nosso código ser mais limpo, legível, simples e organizado, porém o JavaScript também possui regras que devemos seguir. Com relação às variáveis, utilizamos elas através de nomes simbólicos para armazenar os valores / informações dentro dos nossos programas.

O nome das variáveis, são chamados de **identificadores** e precisam obedecer determinadas regras na sua criação, para não gerar erros no seu programa.

E uma boa prática é relacionar o nome de sua variável com o que ela irá armazenar. Um exemplo simples é se queremos armazenar nome completo de uma pessoa, podemos criar uma variável como:

**let nomeCompleto = ‘Daniel Porto’;**

Mas fique tranquilo, iremos mostrar o que pode ou não pode ao declararmos o identificador de uma variável!

# Módulo 2 - Dados, Variáveis e Operações - Declarando e Definindo Variáveis (2 / 7)

## 1 Começar com uma letra

Um identificador Javascript pode começar com qualquer letra de “a” a “z”.E não pode haver nomes duplicados, ou seja, duas variáveis chamadas com o mesmo nome.

```
JS declaracaoVariaveis.js ●   regrasNomenclaturaVariavel.md
JS declaracaoVariaveis.js > $variavelTambemValida
1   let $variavelTambemValida        CERTO
```

O VS Code identifica se houver um erro no nome da sua variável, ficando com um sublinhado vermelho.

```
JS declaracaoVariaveis.js 3 ●   regrasNomenclaturaVariavel.md
JS declaracaoVariaveis.js
1   let 1variavelTambemValida        ERRADO
```

## 2 Podem iniciar com cifrão($) ou underline(_)

Não podemos utilizar caracteres especiais ou termos espaços no identificador. Apenas é permitido o **$** e o **_**.

## 3 Não podem começar com números

Não podemos iniciar os identificadores com números. Porém, podemos utilizar números ao longo do seu nome.

## 4 Não podem ser palavras reservadas

Não podemos utilizar palavras reservadas do JavaScript, são aquelas que já fazem parte da sintaxe da linguagem Javascript.

## 5 Padrão camelCase

Devido ao JavaScript ser case-sensitive, respeitamos o padrão camelCase, onde intercalamos palavras minúsculas com maiúsculas, nessa ordem.

# Módulo 2 - Dados, Variáveis e Operações - Declarando e Definindo Variáveis (3 / 7)

1

## Nomes iniciando com letra

```
1  let soma
2  const registroNacional
3  let variavelSomatorio
```

2

## Nomes com $ e _

```
Exemplos de identificadores permitidos: $caixinha , variavelSomatorio

Não permitido: @transgressora, variavel-errada
```

3

## Nome com números

```
Exemplos corretos: valorProduto1, temTracao4Rodas

Exemplo incorreto: 1NomeErrado
```

4

## Exemplos de Palavra Reservada

```
Exemplos não permitidos: let, const, function, if
```

5

## Exemplos de Padrão camelCase

```
Exemplos: javascriptImpressionador, variavelCriadaParaEsseExemplo,
desenvolvimentoComJavascript, programacaoEhLegal
```

O padrão **camelCase** é uma convenção de nomenclatura utilizada em JavaScript e algumas outras linguagens, para facilitar a padronização e compreensão na leitura de códigos. O seu padrão é escrever palavras compostas ou frases, onde há primeira palavra é iniciada com letra minúscula e é intercalada com palavras de letra maiúscula e unidas sem espaços.

O nome camelCase é dado devido a aparência das costas de um camelo (camel em inglês).

Outro padrão que podemos utilizar é o **snakeCase** que é um estilo de escrita em que cada espaço é substituído por um sublinhado(_) e a primeira letra de cada palavra é escrita de forma minúscula (lower_snake), é utilizada convencionalmente para declarar nomes de **campos de bancos de dados, variáveis e funções**.

Existe a padronização com letras maiúsculas que é denominada **CAPA_SNAKE** ( upper_snake) que é frequentemente utilizada como convenção para declarar **constantes**.

Além de algumas conveções como **kebab-case** (dash-case) que os espaços são substituídos com hífen(-), **PascalCase** que combina palavras colocando todas em letra maiúsculas, inclusive a primeira.

⚠

**O padrão que iremos utilizar em JavaScript será o camelCase**

## Tab (auto-completar)

Ao começar digitar um comando no terminal, como nome de arquivo, diretório ou até mesmo opções de comando. Ao apertar a tecla tab ela irá automaticamente completar o que vocês estava digitando ou mostrará todos os resultados possíveis.

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   SQL CONSOLE
PS C:\Users\danie\JavaScript Impressionador\modulo_2> node .\dec
```

**Aperte tecla TAB**

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   SQL CONSOLE
PS C:\Users\danie\JavaScript Impressionador\modulo_2> node .\declaracaoVariaveis.js
```

# Módulo 2 - Dados, Variáveis e Operações - Declarando e Definindo Variáveis (6 / 7)

```
JS declaracaoVariaveis.js 1 ×   regrasNomenclaturaVariavel.md
JS declaracaoVa
1   let va
2   variav
3
4   const ultimaVariavelDaAula;
5
6   ultimaVariavelDaAula = 'finalmente coloquei um valor nessa variável';
7
8   console.log(variavelNova);
9   console.log(ultimaVariavelDaAula);
```

```
const ultimaVariavelDaAula: any
'const' declarations must be initialized. ts(1155)
View Problem (Alt+F8)   No quick fixes available
```

```
JS declaracaoVariaveis.js ×   regrasNomenclaturaVariavel.md
JS declaracaoVariaveis.js > ...
1   const variavelNova = 'valorDessaVariavel';
2   variavelNova = 'novoValor';
3
4   console.log(variavelNova);
```

## Constantes

Recapitulando sobre as variáveis do tipo constantes, as regras de nomenclatura irão ser aplicadas a este tipo, mas lembre-se que uma constante precisa ser **inicializada** , ou seja, na hora de criarmos uma variável do tipo const, devemos atribuir o valor que queremos que ela guarde, e esta informação não poderá ser alterada futuramente, e ao executar seu código irá gerar erros de execução.

## Erros de execução

Ao executarmos no terminal um código com erro de declaração de constante , ele irá apontar o erro gerado, tipo e o local onde está o erro.

```
PS C:\Users\danie\JavaScript Impressionador\modulo_2> node .\declaracaoVariaveis.js
C:\Users\danie\JavaScript Impressionador\modulo_2\declaracaoVariaveis.js:2
variavelNova = 'novoValor';
             ^

TypeError: Assignment to constant variable.
    at Object.<anonymous> (C:\Users\danie\JavaScript Impressionador\modulo_2\declaracaoVariaveis.js:2:14)
    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
```

Já a estrutura da variável do tipo let, nos permite tanto reatribuir valores para as variáveis já criadas, com também permite inicializar a variável sem atribuir um valor inicialmente:

```
let nomeVariavel;
```

O formato acima é a forma de declararmos uma variável sem atribuir um valor a ela.

E quando precisarmos dessa variável, iremos atribuir um valor a ela, em linhas futuras do nosso código, dessa forma:

```
nomeVariavel = ' Estou recebendo meu primeiro valor, pois sou uma LET';
```

Exemplos de declarações de variáveis tipo let e atribuições de valores:

JS declaracaoVariaveis.js × regrasNomenclaturaVariavel.md

JS declaracaoVariaveis.js > ...

```
1  let variavelNova = 'valorDessaVariavel';
2  variavelNova = 'novoValor';
3
4  let ultimaVariavelDaAula;
5
6  ultimaVariavelDaAula = 'finalmente coloquei um valor nessa variável';
7
8  console.log(variavelNova);
9  console.log(ultimaVariavelDaAula);
```

Saída do código acima ao ser executado:

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL SQL CONSOLE

```
PS C:\Users\danie\JavaScript Impressionador\modulo_2> node .\declaracaoVariaveis.js
novoValor
PS C:\Users\danie\JavaScript Impressionador\modulo_2> node .\declaracaoVariaveis.js
novoValor
finalmente coloquei um valor nessa variável
PS C:\Users\danie\JavaScript Impressionador\modulo_2>
```

**O que são operadores JavaScript?**

Os **OPERADORES** são símbolos especiais que envolvem um ou mais operandos com a finalidade de produzir um determinado resultado, ou seja, símbolos que sinalizam ao nosso interpretador JavaScript a operação que desejamos realizar com os nossos dados. Pense em uma equação de matemática simples como a Adição, quando escrevemos 2 + 3 , esperamos que o **operador +** realize a soma de 2 com 3, gerando um resultado 5, correto?

Adição: 2 + 3 = 5

Os nossos operadores funcionarão da mesma maneira, recebendo uma tarefa para fornecer um resultado. Existem alguns tipos de operadores em JavaScript, no caso da **Adição (+)**, ela corresponde a um tipo de operador chamado **ARITMÉTICO**. E vamos falar mais um pouco sobre eles a seguir.

## Operadores Aritméticos

Esse operadores podem ser velhos conhecidos das aulas de matemática. São usados para fazer operações com valores numéricos (sejam literais ou variáveis) como seus operandos e retornam um único valor numérico.

Na tabela ao lado, identificamos alguns operadores padrões como adição, subtração, multiplicação e divisão. Mas existem outros que complementam o nosso quadro de operadores, como a exponenciação, resto da divisão, incremento e decremento.

| Descrição | Operador | Exemplo |
|---|---|---|
| ADIÇÃO | + | 2 + 8 |
| SUBTRAÇÃO | - | 2 - 8 |
| MULTIPLICAÇÃO | * | 2 * 8 |
| DIVISÃO | / | 2 / 8 |
| EXPONENCIAÇÃO (OU POTENCIALIZAÇÃO) | ** | 2 ** 8 |
| RESTO DA DIVISÃO (OU MODULUS) | % | 2 % 8 |
| INCREMENTO | ++ | 1++ |
| DECREMENTO | -- | 1-- |

2

O operador de **Exponenciação (ou Potencialização)** calcula a base elevada à potência do expoente, ou seja, na matemática falávamos o 'número elevado a 3' (ao cubo).

$2^3$ → **2**3**

O operador **Modulus (Módulo) ou Resto da divisão** retorna o inteiro restante da divisão dos dois operandos.

**12 % 5 retorna resto 2**

**10 % 5 retorna resto 0**

3

Os operadores de **Incremento ( ++ ) e Decremento ( -- )** diferente dos outros operadores precisam apenas de um número ou uma variável para realizar sua operação.

O Incremento adiciona um ( 1 ) ao seu operando e o Decremento subtrai um ( 1 ) ao seu operando.

**let x = 10;**
**x++**
**Retorna x = 11**

**let x = 10;**
**x--**
**Retorna x = 9**

# Módulo 2 – Dados, Variáveis e Operações - Operadores Aritméticos (4/ 8)

Porém você pode estar se perguntando, será que os Operadores Aritméticos são utilizados apenas com valores numéricos?

A resposta é **não**, existem algumas exceções, como o operador de Adição ( + ) que conseguimos utilizar como **operador de textos**.

Esse operador é chamado de **CONCATENAÇÃO** é utilizado para unir / juntar textos em JavaScript.

O resultado que ele produz pode ser visto como uma soma da matemática, mas como funciona esse processo?

operadores.js × operadoresJavascript.md

aula4 > operadores.js > ...

```
1  const texto1 = 'Oi, meu nome é Daniel! ';
2  const texto2 = 'Eu tenho 29 anos.';
3
4  const textoFinal = texto1 + texto2;
5  console.log(textoFinal);
```

```
PS C:\Users\danie\JavaScript Impressionador\modulo_2> node ./aula4/operadores.js
Oi, meu nome é Daniel! Eu tenho 29 anos.
```

O **operador de Adição ( + ) avalia os valores** dos operandos e retorna um novo texto que é a junção desses valores. Vale lembrar que neste caso não precisam necessariamente ser todos do tipo texto, pois o JavaScript tentará transformar os valores para fazer a concatenação.

```
1  const texto1 = 'Oi, meu nome é Daniel! ';
2  const texto2 = 'Minha idade é ';
3  const numeroIdade = 29;
4
5  const textoFinal = texto1 + texto2 + numeroIdade;
6  console.log(textoFinal);
```

```
PS C:\Users\daniel\JavaScript Impressionador\modulo_2> node ./aula4/operadores.js
Oi, meu nome é Daniel! Minha idade é 29
```

```
1   const texto1 = 'Oi, meu nome é Daniel! ';
2   const texto2 = 'Eu tenho 29 anos.';
3
4   const textoFinal = texto1 + texto2;
5
6
7   const variavelNumerica1 = 27;
8   const variavelNumerica2 = 35;
9
10  const resultadoMultiplicacao = variavelNumerica1 * variavelNumerica2;
11  console.log(textoFinal);
12  console.log(resultadoMultiplicacao);
```

Com os nossos operadores aritméticos as operações podem ser feitas diretamente dentro de uma variável ou entre variáveis, como nos exemplos acima, em que cada variável armazena um valor, seja numérico ou texto, e são realizadas as operações entre elas. E irão retornar o resultado das operações.

```
PS C:\Users\danie\JavaScript Impressionador\modulo_2> node ./aula4/operadores.js
Oi, meu nome é Daniel! Eu tenho 29 anos.
945
```

Mas precisamos tomar cuidado, pois nem todos os operadores podem ser utilizados em casos que não sejam valores numéricos. Como a Subtração ( - ) que se colocado no exemplo anterior, irá gerar um resultado do tipo **NaN**.

**NaN** é um acrômio em inglês para **Not A Number**, ou seja, Não é um número. É um valor que representa um valor numérico indefinido ou irrepresentável.

No exemplo ao lado, estamos tentando subtrair um texto de outro, mas o JavaScript ao ler os valores, verifica que tal operação não é possível, porém ele não irá lançar um erro nesse caso, mas sim dará um retorno indicando que não foi possível após avaliar, transformar esses valores em texto.

aula4 > JS operadores.js > [@] textoFinal

```
1  const texto1 = 'Oi, meu nome é Daniel! ';
2  const texto2 = 'Eu tenho 29 anos.';
3
4  const textoFinal = texto1 - texto2;
5  console.log(textoFinal);
```

```
PS C:\Users\danie\JavaScript Impressionador\modulo_2> node ./aula4/operadores.js
NaN
```

E se eu contar que existe um tipo de operador que você vem utilizando desde do início das nossas aulas??

Isso mesmo que você leu!

O nosso **Operador de Atribuição**, o símbolo de **igual ( = ).** Utilizamos ele para atribuir um valor a uma variável.

```
1   const primeiraVariavel = 'essa é a minha variavel';
```

Independentemente do tipo de declaração de variável que escolhermos utilizar (let ou const), após nomearmos a nossa variável, utilizamos nosso **operador de atribuição =** para armazenar um novo valor ou substituir o valor original da variável.

```
PS C:\Users\danie\JavaScript Impressionador\modulo_2> node ./aula4/operadores.js
```

No caso acima, o arquivo que queremos executar **operadores.js**, está dentro da pasta **aula4**, que está dentro da pasta **modulo_2** (indicado pelo **./** ).

## Executando arquivos

Caso você esteja em outra pasta, ao invés da pasta que está o seu arquivo código, é só passar o caminho com os nomes do diretórios até chegar aonde seu programa está após o comando node. E o seu programa será executado de forma e no local correto.

Nesse momento, iremos revisitar alguns conceitos para fortalecer a compreensão deles. Vamos então retornar um pouco sobre o tema **variáveis.**

Variáveis são "caixas" nas quais iremos armazenar informações ou valores, e ao declararmos elas, utilizamos os tipos **Const** ou **Let.** A escolha para cada tipo será de acordo com a sua mutabilidade, ou seja, se queremos que seja uma variável do tipo constante, aquela que não terá seu valor alterado, utilizamos o tipo **const**, ou se queremos alterar o seu valor ao longo do nosso programa, utilizamos o tipo **let**.

Após a declaração do tipo, precisamos nomear essa variável, como se colocássemos uma "etiqueta" na nossa caixa, para conseguir utilizar, invocar, chamar ou alterar ela no nosso programa.

Utilizamos o operador de atribuição **igual ( = )** , para atribuir um dado ou valor para essa variável, como exemplo um texto ( sempre entre **aspas simples ' '** ou **duplas " "**.

Exemplo de declaração de uma variável

```
JS aula5.js > ...
1   let variavelCaixinha1 = "Esse é o meu primeiro texto do arquivo!";
```

Independentemente do tipo de variável que criarmos ou utilizarmos, elas não podem ter a mesma nomenclatura, ou seja, não pode existir variáveis com o mesmo nome, o próprio VS Code irá mostrar o erro.

```
JS aula5.js > [@] variavelCaixinha1
1   let variavelCaixinha1 = "Esse é o meu primeiro texto do arquivo!";
2   const variavelCaixinha1
```

Outro erro comum, e que também o VS Code irá mostrar, é quando não inicializamos uma variável do tipo const, ou seja, quando não atribuímos um valor no momento em que criamos a variável.

```
const variavelCaixinha2
```

Então **SEMPRE** que criarmos uma variável **constante**, precisamos atribuir o seu valor, porque se ela não inicializar com um valor na mesma linha que sua criação, ela nunca teria um valor atribuído devido a sua característica de ser imutável.

Agora criarmos uma variável do tipo let e não atribuirmos nenhum valor inicial, não teria problema, é como se estivéssemos preparando a nossa "caixinha" / variável para receber algum valor futuramente, pois o **tipo let é mutável**, e nos permite alterar o seu valor original.

Aprendemos que o JavaScript recebe instruções, em forma de comandos, para que execute uma tarefa. Um comando que utilizamos bastante até agora é o **console.log()**. Ele é o comando que permite imprimir os valores, que são passados entre seus parênteses ( ), na sua tela / terminal de texto.

```
JS aula5.js > ...
1  let variavelCaixinha1 = "Esse é o meu primeiro texto do arquivo!";
2  const variavelCaixinha2 = 'aqui se encontra o meu segundo texto'
3
4  console.log(variavelCaixinha1);
5  console.log(variavelCaixinha2)
```

**Instrução**

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   SQL CONSOLE

PS C:\Users\danie\JavaScript Impressionador\modulo_2\aula5> node ./aula5.js
Esse é o meu primeiro texto do arquivo!
aqui se encontra o meu segundo texto
```

**Saída Terminal**

Agora vamos analisar um pouco mais de perto o código acima. Você percebeu que utilizamos **aspas simples ' '** em uma variável, depois **duplas " "**. Uma hora colocamos o **ponto e vírgula ( ; )**, outros momentos não. E vimos que o nosso programa funcionou sem nenhum erro, resultando na saída esperada do nosso console.log().

Sabemos que para utilizar textos, podemos utilizar os dois tipos de aspas, e que o nosso ponto e vírgula é opcional.

**Mas então qual é o Certo e Errado??**

O JavaScript é muito flexível, em termos de algumas regrinhas, como o ponto e vírgula e as aspas. Ao rodarmos um programa em JavaScript o seu interpretador, consegue identificar quando começa e termina uma linha de código, que tanto as aspas simples como duplas são utilizados para informações do tipo texto. Então para ele, tudo estará certo, o importante é seguir um **PADRÃO**.

Então para entender como escrever um código com boa qualidade, vamos falar sobre **PADRONIZAÇÃO DE CÓDIGO**.

A Padronização de código em JavaScript é manter a **consistência do estilo de código** que está escrevendo, pensando que futuramente você precise revisitar o programa que criou, ou até mesmo criar um projeto em Pair Programming (você com mais pessoas), o que é muito comum na vida diária de um programador, para isso vocês precisam ter uma comunicação única em termos de código.

As vantagens da padronização é a **Legibilidade** do código, a fácil **Manutenção**, **Uniformidade** e melhores práticas ao desenvolver seu programa, o que aumenta a qualidade do código.

# Módulo 2 – Dados, Variáveis e Operações - Padronizando o Código (5/ 8)

Para facilitar a padronização o VS Code, possui uma ferramenta chamada Extensões, onde conseguimos baixar facilitadores de código, legibilidade, formatação, auto preenchimento, entre muitos outros.

Hoje vamos ver a **Extensão Prettier - Code formatter**, que é uma extensão de formatação de código, ele permite você deixar o seu código em um formato pré-estabelecido, ou seja, em um padrão.

1. Clique no ícone 'Extensions', do lado direito do VS Code.
2. Digite no campo de busca: Prettier ( o primeiro já é a extensão que queremos).
3. Clique no primeiro resultado, e clique em instalar (caso ainda não tenha).

## Formatação do código

Para utilizar a **Extensão Prettier - Code formatter**, O VS Code possui um atalho – Alt + Shift + F, que é utilizado para formatação do código e a extensão torna o atalho mais versátil e mais funcional para a linguagem JavaScript.

aula5.js > ...

```
1  let variavelCaixinha1 = "Esse é o meu primeiro texto do arquivo!";
2  const variavelCaixinha2 = 'aqui se encontra o meu segundo texto'
3
4  console.log(variavelCaixinha1);
5  console.log(variavelCaixinha2)
```

**ALT + SHIFT + F**

```
1  let variavelCaixinha1 = 'Esse é o meu primeiro texto do arquivo!';
2  const variavelCaixinha2 = 'aqui se encontra o meu segundo texto';
3
4  console.log(variavelCaixinha1);
5  console.log(variavelCaixinha2);
```

# Módulo 2 – Dados, Variáveis e Operações - Padronizando o Código (7/ 8)

Para você acessar as **configurações do VS Code** e verificar qual padronização está sendo utilizada ou configurar o padrão que você quer que o VS Code formate o seu código, você irá acessar através do atalho ' **CTRL + vírgula ( , )** ' . Dentro das configurações, clicando em E**xtensões**, toda a lista de extensões salvas no seu VS Code irá aparecer.

Nesse momento iremos localizar o **Prettier** e clicar nele mostrando ao lado direito todas as configurações e opções que a extensão possui. Não se preocupe se não sabe tudo o que está ali, ou em quais opções você deve mexer, o Prettier já vem como padrão algumas configurações básicas, e ao longo de sua jornada ficará mais fácil entender qual é o padrão de código que irá seguir ( sabendo que será diferente dependendo do ambiente de trabalho ou projeto).

Uma recomendação para facilitar a formatação do seu código, é deixar como padrão a formatação no momento em que salvar o arquivo. Isso facilita o uso da ferramenta, além de economizar tempo e não correr o risco de esquecer de formatar o seu código.

Para isso, ainda nas configurações do VS Code, procuraremos por ' **format on save**' na barra de busca, é só verificar se a configuração está ativa, senão basta clicar no **seletor Format On Save** e pronto! Toda vez que **salvar** o seu código (**CTRL + S**), o VS Code irá formatar.

Anteriormente iniciamos o conceito de **Variáveis** em JavaScript e como realizar suas declarações e criações, e utilizamos bastante o comando console.log() para visualizarmos elas.

Agora vamos entender um pouco mais sobre os Tipos de Dados que o JavaScript possui. E para isso precisamos entender o que é um **DADO.**

Um **DADO**, em qualquer linguagem de programação, é uma unidade ou elemento de informação que pode ser acessado através de algum identificador. No caso das variáveis, o que elas guardam ( um dado/ uma informação ) que poderá ser acessada quando utilizarmos essas variáveis.

Em JavaScript um Dado é a **palavra de comunicação,** onde ele irá processar o que está no dentro dele e realizar alguma coisa. Alguns dados você já utilizou nas aulas anteriores que são os dados de textos e números. Eles pertencem a tipos de dados diferentes e vamos entender como eles funcionam a seguir.

**TIPO STRING**

Trata-se do tipo de dado que são sequências de caracteres alfanuméricos (letras, números e/ou símbolos), ou seja, **Strings** são úteis para guardar dados que podem ser representados em forma de texto. Em JavaScript, uma String sempre estará entre **aspas simples ( ' )** , **aspas duplas ( " )** ou entre **crases ( ` )**.

Até o momento, você percebeu que apenas utilizamos atribuições de variáveis com conteúdo textual com as aspas, e sabemos que não existe diferença para o JavaScript em utilizar aspas simples ou duplas e o retorno do nosso conteúdo será igual, mas o importante é manter um padrão de formatação do seu código.

```
const texto1 = 'Aqui temos uma string normal.';
const texto2 =
  'Já essa, apesar das aspas duplas ao invés das simples, também é uma string normal';

const numeroQueQueroRegistrar = 31472;
```

Agora, a utilização de crases ( ` ) muda um pouco o cenário da string. Quando utilizamos a crase para envolver nossa String, estamos criando uma **string template ( ou template literals)**, que é uma feature do ES6 ( JavaScript mais atual) que permite criar strings complexas mais facilmente,pois ela possui algumas funcionalidades que simplificam essas strings.

**String Template ( Template Literals)**

Vamos lembrar um pouco sobre a **Concatenação de strings e variáveis** em JavaScript, que era como estávamos unindo textos diferentes:

```
JS operadores.js ×   operadores/javascript.md
aula4 > JS operadores.js > ...
1  const texto1 = 'Oi, meu nome é Daniel! ';
2  const texto2 = 'Eu tenho 29 anos.';
3
4  const textoFinal = texto1 + texto2;
5  console.log(textoFinal);
```

```
PS C:\Users\danie\JavaScript Impressionador\modulo_2> node ./aula4/operadores.js
Oi, meu nome é Daniel! Eu tenho 29 anos.
```

O formato de concatenar não irá apresentar erros na execução do seu código. Mas imagine o cenário no qual você esqueça de colocar um espaço ou pontuação em algumas das frases e ela ter um retorno nesse formato: **Oi, meu nome é DanielEu Tenho 29 anos.**

O **String Template**, facilita esse processo de união de strings, de uma forma mais simples as strings mais complexas. A sintaxe pede o uso do sinal de crases (`) no envolto da frase e ao utilizarmos o conteúdo de variáveis colocamos o nome dela dentro de ${ }.

```
const numeroQueQueroRegistrar = 31472;
const stringTemplate = `O número que decidi registrar foi o ${numeroQueQueroRegistrar}`;

//console.log(stringTemplate) fará aparecer na tela o seguinte texto:
// O número que decidi registrar foi o 31472
```

O formato da **String Template** também pode ser utilizado para adicionais expressão dentro das chaves, como **${a + b}.**

```
JS strings.js > ...
1   const a = 10;
2   const b = 20;
3   const stringTemplate = `O resultado da soma é ${a+b} `;
4
5   console.log(stringTemplate);
```

```
O resultado da soma é 30
```

Você consegue adicionar quebras de linhas ( colocar textos em diferentes linhas – um em cima do outro) sem a necessidade de concatená-las com o Operador **+** e **/n** (codificação de quebra de linha).

```
JS strings.js > ...
1   const nome = 'Daniel';
2   const idade = 29;
3   const stringTemplate = `Oi, meu nome é ${nome}!
4   Minha idade é ${idade}`;
5
6   console.log(stringTemplate);
```

```
Oi, meu nome é Daniel!
Minha idade é 29
```

**TIPO NUMBER (Número)**

Trata-se do tipo de dado que representa **informações numéricas**. Sejam números inteiros positivos e negativos (-3, 0 ,1, 12, 187...) ou números decimais (23.456), todos são entendidos como dado do **tipo Number**.

Uma curiosidade sobre o tipo Number no JavaScript, é que todo número é um tipo Number, diferente de outras linguagens de programação. Por exemplo, os números decimais, são conhecidos como números de pontos flutuantes (ou **floats**) e eles não seriam caracterizados como tipo Number, mas no caso do JavaScript não.

Vamos aprender ao longo do curso, que os dados possuem **propriedades e métodos** para trabalharmos ao longo do nosso código, e essa informação é importante, pois independentemente do número, ele possuirá as propriedades que existem no tipo Number.

```
const numero1 = 27;
const numero2 = 23.25;
const numero3 = -40;
const total = numero1 + numero2 + numero3;

console.log(total);

//10.25
```

Ao falarmos de Dados, precisamos entender primeiro como um computador irá processá-los. E para isso, vamos falar sobre os conceitos de **Números Binários e Bits**.

O **computador** sendo inspecionado bem a fundo, pode ser entendido como nada mais de que **circuitos elétricos**, ou seja, uma placa eletrônica onde passam vários caminhos de corrente elétrica, podemos entender que o computador nada mais é que um desses circuitos, onde a **inteligência do computador é processar e gerenciar esses caminhos de energia** (pode passar energia ou não).

Para gerenciar esse caminho, o computador utiliza apenas dois números para processar esse funcionamento. São o **número 0**, que indica que **não há energia/corrente** e o **número 1**, que indica que **há sim energia/corrente**. Essa informação binária (descrita por 0 ou 1) é chamada de **BIT**.

**Bit** é a menor unidade de informação que o computador pode utilizar para trabalhar. Pode assumir somente dois valores: 0 ou 1 ( verdadeiro ou falso).

**Então como o Computador consegue trabalhar com números maiores do que 1?**

Para descrevermos números maiores, precisamos agrupar os Bits, ou seja, um Bit significa 0 ou 1, dois Bits me dão mais possibilidades gerando mais números :

O uso de dois algarismos binários, conseguimos descrever 4 números diferentes ( 0 a 3).

```
00 => número 0
01 => número 1
10 => número 2
11 => número 3
```

Isso acontece porque basicamente estamos somando os únicos dois valores que um Bit pode ter, vejamos o exemplo do 2:

```
 1
01
01 +
____
10
```

É como se quisemos somar os números 1, e para isso aumentamos a casa da dezena dele, como quando fazemos a soma 7 + 9 = 16, onde possuímos dois números na unidade e ao somar, colocamos um número na casa da dezena.

É uma representação diferente do número 2, pois é uma representação binária dele, porém 1 + 1 continua sendo 2.

Vamos perceber que existe um padrão para representação binária dos números, onde o agrupamento de mais Bits descreve mais números. Para saber quantos números conseguiremos descrever com o agrupamento, basta elevar o número 2 com o número de Bits que deseja agrupar e o resultado é a quantidade de números que conseguimos representar :

- com 2 bits, tivemos 4 números diferentes => 4 = $2^2$
- com 3 bits, tivemos 8 números diferentes => 8 = $2^3$
- com 4 bits, tivemos 16 números diferentes => 16 = $2^4$
- com 5 bits, tivemos 32 números diferentes => 32 = $2^5$
- **com n bits, teremos $2^n$ números diferentes**

Uma forma simplificada de entendermos a representação binária é que a soma de 1 + 1 = 0 e 0 + 1 = 1. Então quando falamos de um agrupamento de 3 bits, iremos representar os números da seguinte forma:

```
  1 1
  0 1 1
 +0 0 1
 -------
  1 0 0
    =
    4
```

```
000 -> 0
001 -> 1
010 -> 2
011 -> 3
100 -> 4
101 -> 5
110 -> 6
111 -> 7
```

```
  1
  0 1 1
 +0 1 0
 -------
  1 0 1
    =
    5
```

Número 7 é a representação máxima de 3 bits.
Precisamos inserir um 4° bit para descrever o número 8.

A partir da junção de 8 bits, foi criado uma nova unidade para facilitar o processo, chamado **BYTE.**

Então um **BYTE** representa $2^8$ **= 256** números**.**

Agora para representarmos números negativos, podemos dedicar 1 bit para o sinal, ou seja, em um agrupamento de 3 bits, podemos dizer que 1 bit será o sinal, e os outros 2 bits irão representar os números, nesse caso ( -1, -2, -3).

111 sendo o número -3
011 sendo o número +3

Anteriormente vimos que o computador entende o **número 0 e o 1**, como a capacidade de verificar se há ou não há energia no caminho do circuito, e que também conseguimos representar diversos números através de **agrupamentos de 0 e 1** (**bits e bytes**), porém o computador não apenas possui a capacidade de representar números através do 0 e 1 , como também **qualquer tipo de dado** que ele esteja processando, como letras e símbolos.

Esse tipo de **linguagem** é chamada de **baixo nível**, pois são voltadas ao **entendimento da máquina** e possuem instruções mais diretas para o processador. Facilitando no processamento das informações, ou seja, gasta menos energia para ser compreendida.

Existe também a **linguagem de alto nível,** que possui a sintaxe voltada para o **entendimento humano**, ela é mais intuitiva, amigável e fácil de aprender, isso ocorre porque a linguagem de alto nível abstrai conceitos voltados para máquinas e sintetizam em instruções ou comandos como utilizamos na programação. E por exigir uma tradução do computador, seu processamento é mais lento.

**O JavaScript é uma linguagem de alto nível**, justamente por ter seus comandos mais próximos do que escrevemos e consequentemente de mais fácil entendimento. O processo de escrever o código, de programar, está mais próximo de nós humanos.

Dessa forma o computador irá processar os nossos comandos, traduzir para o código binário e entender esses zeros (0) e uns (1) como a linguagem dele e executar aquilo que foi instruído.

**TIPO BOOLEAN**

Dentre os tipos de dados que o JavaScript possui o **tipo BOOLEAN** utiliza o conceito dos números binários, de **só aceitar dois valores**. Mas como mencionamos que o JavaScript é uma linguagem que se aproxima com a linguagem humana, não faz sentido utilizarmos os valores 0 e 1, pois são de entendimento da máquina, ou seja, o JavaScript utiliza de uma "roupagem mais humana", ao invés de 0 e 1, os valores que o **tipo Boolean** permite são os valores **TRUE (verdadeiro)** e **FALSE (falso)**.

Uma variável com esse tipo de informação atribuída são chamadas de **VARIÁVEIS BOOLEANAS** ou do **tipo boolean**.

```
const valor1 = false;
const valor2 = true;
```

Repare que não colocamos as palavras **false** e **true** entre aspas ou crase, pois **NÃO** são textos e sim **valores específicos** que uma variável booleana armazena.

## Operadores Booleanos ou Lógicos

Variáveis booleanas possuem seus próprios **operadores Booleanos** ou **operadores lógicos**.

São operadores importantes no JavaScript que permitem comparar variáveis, seu **resultado** sempre será **verdadeiro (true)** ou **falso (false)**.

E é através do resultado dessa comparação que iremos realizar algo no nosso programa.

| Operador | Nome da Operação | Descrição |
|---|---|---|
| **&&** | **AND** ( operação **e** ) | Usamos essa operação quando desejamos que DUAS condições sejam SIMULTANEAMENTE verdadeiras. |
| **\|\|** | **OR** ( operação **ou** ) | Usamos essa operação quando desejamos que pelo menos UMA das condições seja verdadeira. |
| **!** ( negação ) | **NOT** ( Inversor ) | Esse é um operador aplicado a UM VALOR booleano quando queremos o seu INVERSO (NÃO É algo). |

Vamos analisar alguns exemplos de uso dos operadores lógicos:

```
const valor1 = true;
const valor2 = false;

const valor1EValor2 = valor1 && valor2;
const valor1OUValor2 = valor1 || valor2;
const inversoValor1 = !valor1;
const inversoValor2 = !valor2;

console.log(valor1EValor2);
//false
console.log(valor1OUValor2);
//true
console.log(inversoValor1);
//false
console.log(inversoValor2);
//true
```

Ao lado declaramos duas variáveis booleanas, cada uma com um valor (true e false). Logo após atribuímos os resultados das comparações entre a variável valor1 e valor 2 em novas variáveis. Utilizando os operadores **AND (&&), OR( || )** e **NOT ( ! )**, temos os seguintes retornos:

Quando utilizamos o **operador AND** comparamos se os valores das duas variáveis possuem as mesmas condições, nesse exemplo um é Verdadeiro e o outro é Falso, então o retorno será FALSO. Apenas seria verdadeiro, se as duas informações fossem true ou as duas fossem false.

Agora quando utilizamos o operador **OR**, o retorno se torna VERDADEIRO, já que uma de suas condições é true.

E o **operador NOT** sempre inverterá o valor que a variável possui, valor1 se retorna false, e valor2 se retorna true.

## Operadores de Comparação

Os operadores de comparação em JavaScript são utilizados para comparar dois valores e retornar um valor lógico (verdadeiro ou falso) com base na comparação.

Esses operadores são amplamente utilizados em estruturas condicionais, como o if, para avaliar se uma determinada condição é verdadeira ou falsa.

Existem vários operadores de comparação em JavaScript, cada um com sua própria função e sintaxe.

| Descrição | Operador | Exemplo |
|---|---|---|
| MAIOR QUE | > | a > b |
| MENOR QUE | < | a < b |
| MAIOR OU IGUAL QUE | >= | a >= b |
| MENOR OU IGUAL QUE | <= | a <= b |
| IGUAL A | == | a == b |
| IGUALDADE ESTRITA (IGUAL A TIPO E VALOR) | === | a === b |
| DIFERENTE DE | != | a != b |

Vamos executar alguns exemplo e observar o comportamento dos operadores de comparação, lembrando que seu retorno sempre será um booleano.

No exemplo abaixo, iniciamos uma variável x e atribuímos o valor 2. Em seguida, são feitas duas comparações utilizando os **operadores de comparação > (maior que)**.

Na primeira comparação **x > 1**, o valor de x é de fato maior que 1, portanto, a expressão retorna **true.**
Na segunda comparação **x > 10,** o valor de x é menor que 10, portanto, a expressão retorna **false.**

JS comparadores.js > ...

```
1   let x = 2;
2
3   console.log(x > 1);
4   console.log(x > 10);
```

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_2\aula10> node .\comparadores.js
true
false
```

É importante lembrar que os operadores de comparação também podem ser utilizados com outros tipos de dados, como strings, booleanos e objetos.

A **comparação entre strings** é feita com base na ordenação lexicográfica dos caracteres. Já a comparação entre objetos é feita com base na referência de memória, ou seja, dois objetos só serão considerados iguais se forem exatamente o mesmo objeto.

Vamos a outro exemplo:

```
let y = "string de teste";
console.log(y === "string de teste");
console.log(y === "novo texto que eu inventei");
```

A primeira comparação y === "string de teste" retorna true porque a string atribuída a y é igual à string sendo comparada.

Já a segunda comparação y === "novo texto que eu inventei" retorna false porque a string atribuída a y é diferente da string sendo comparada.

Isso indica que a primeira comparação retorna true, pois as duas strings são iguais, enquanto a segunda comparação retorna false, pois as strings são diferentes.

É importante notar que o operador de igualdade estrita === compara tanto o valor quanto o tipo dos operandos. Nesse caso, as duas strings são do tipo string e têm conteúdos diferentes, resultando em uma comparação falsa.

É importante ressaltar que o **operador ==** realiza uma comparação de igualdade não estrita, o que significa que ele faz uma conversão de tipos antes de realizar a comparação. Por exemplo, se compararmos o número 1 com a string "1", o **operador ==** retornará true, pois ele converte a string em um número antes de realizar a comparação.

Por outro lado, o **operador ===** realiza uma comparação de igualdade estrita, levando em consideração tanto o valor quanto o tipo dos operandos. Nesse caso, a comparação entre o número 1 e a string "1" retornaria false, pois eles têm tipos diferentes.

Conseguimos comparar os valores para verificar se eles são diferentes:

```
let y = "string de test";
console.log(y !== "novo texto que eu inventei");
```

A variável y é atribuída a string "string de test". Em seguida, é feita uma comparação utilizando o **operador de não igualdade estrita !==** entre y e a string "novo texto que eu inventei".

A comparação y !== "novo texto que eu inventei" retorna true porque a string atribuída a y é diferente da string sendo comparada.
Isso indica que a comparação entre y e a string "novo texto que eu inventei" retorna true, pois as duas strings são diferentes.

É importante ressaltar que o operador de não igualdade estrita !== compara tanto o valor quanto o tipo dos operandos. Nesse caso, as duas strings são do tipo string e têm conteúdos diferentes, resultando em uma comparação verdadeira.

# Módulo 2 – Dados, Variáveis e Operações - Operadores de Comparação (5 / 5)

Conseguimos aplicar diversos tipos de comparação, veja alguns exemplos e seus retornos:

JS comparadores.js > ...

```javascript
let x = 2;
let y = 'string de teste';

console.log(`01:${x > 1}`);
console.log(`02:${x > 10}`);

console.log(`03:${y === 'string de teste'}`);
console.log(`04:${y !== 'novo texto que eu inventei'}`);

console.log(`05:${x >= 2}`);
console.log(`06:${x >= 1}`);
console.log(`07:${x >= 3}`);
console.log(`08:${x <= 3}`);
console.log(`09:${x < 10}`);
```

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_2\aula10> node .\comparadores.js
01:true
02:false
03:true
04:true
05:true
06:true
07:false
08:true
09:true
```

**TIPOS PRIMITIVOS**

Até o momento aprendemos alguns tipos de dados que são classificados como **TIPOS PRIMITIVOS** ( ou tipos "Valor").

Temos no Javascript uma divisão ente os possíveis tipos de dados. De um lado, o conjunto dos tipos Primitivos (ou "Valor"). Do outro lado,o conjunto dos **"Referência"**.

Os tipos que conhecemos até agora são alguns dos tipos primitivos, como **"Number"**, **"String"** e **"Boolean"**. Nessa aula conheceremos outros dois importantíssimos desse primeiro conjunto.

Antes de adentrarmos em cada um dois tipos de dados, vamos compreender a menção do **operador typeof.**

O **typeof** é operador que deve ser acompanhado de algum dado ou variável, e essa combinação produzirá a informação do tipo do dado usado, ou seja, ele mostrará qual o tipo de dado que ele é.

```
4   let valor1 = 2;
5   console.log(valor1);
6   console.log(typeof valor1);
```

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_2\aula11> node ./tipos_valor.js
2
number
```

**TIPO UNDEFINED**

**"Undefined"** (do inglês, "não definido") é um tipo especial de dado no Javascript. Isso porquê **"undefined" é o tipo de dado** e **também o seu valor**, o único tipo dele será **undefined**. Ele é o valor padrão que o Javascript coloca nas variáveis , quando elas não armazenam nenhum valor.

Podemos não declarar conteúdo na variável, então ela será do tipo undefined.

```
let valor1;
console.log(valor1);
// undefined
console.log(typeof valor1);
// undefined
```

Atribuir o valor undefined para uma variável.

```
4   let valor1 = undefined;
5   console.log(valor1);
```

E através do typeof, verificamos o tipo dessa variável.

```
4   let valor1 = undefined;
5   console.log(valor1);
6   console.log(typeof valor1);
```

Verificando o retorno

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_2\aula11> node ./tipoe_valor.js
undefined
undefined
```

**TIPO NULL**

**"Null"** (do inglês, "nulo") é outra representação para o vazio no Javascript. A diferença entre **null** e o **undefined** consiste em sua origem.

O **undefined** é o valor padrão que o Javascript nos entregará quando algo não estiver um valor definido, ao passo que o **null** só se apresentará se alguém **intencionalmente** colocá-lo ali, ou seja, o programador precisa atribuir o seu valor null, para especificar esse tipo.

Atribuímos o valor a uma variável.

```
let valor1 = null;
console.log(valor1);
// null
```

Ao executarmos o **typeof de null** verificamos que é do tipo objeto que seria um dos "Referências", não se sabe, se isso é um bug do Javascript, ou se é intencional ele não ser do tipo null especificamente.

Verificando o retorno

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_2\aula11> node ./tipos_valor.js
null
object
```

## Comentários

Para comentar linhas separadas de código no VS Code, você pode utilizar **//** ou atalho **CTRL+/**.

Para comentar blocos de linhas, é feito com os caracteres **"/*"** no início do trecho e **"*/"** para encerrar o conteúdo comentado, ou o atalho **CTRL+SHIFT+A**.

```
1   // console.log(typeof undefined);
2   // console.log(typeof null);
```

```
1   /* console.log(typeof undefined);
2   console.log(typeof null); */
```

**TIPOS REFERÊNCIA**

Quando falamos de tipos primitivos (ou valor), significa criar uma CÓPIA do original, ou seja, se eu atribuo um valor à variável, e eu for utiliza-la, eu consigo pegar a cópia dessa informação.

Para entendermos o conceito de referência, pensamos em criar um APELIDO (alias) para o original, ou seja, se sua mãe te chama de "Meu anjo", embora seu nome seja Daniel, isso não cria uma cópia de você mesmo, você ainda será você, mas pode ser chamado por esses dois nomes.

E você verá que os tipos estão alinhados com a forma de passarmos valores, e quando eu digo “passar valores”, eu estou me referindo à:

- Atribuir um valor à uma variável;
- Atribuir um valor à uma propriedade ou método de um objeto;
- Passar argumentos para uma função (ou método de um objeto).

A partir de agora veremos 3 tipos de referência : **Objeto**, **Array** e **Função**.

**OBJETO**

Um objeto Javascript trata-se de um "documento", o qual temos campos e valores associados. É uma única variável que contém todo um conjunto de informações sobre algum elemento que se deseja conhecer.

Cada campo do nosso objeto, pode receber tipos de dados diferentes (Number, String, Boolean...).

Para criarmos um objeto precisamos **declarar uma variável**, seu **tipo** e **nome**, atribuir entre chaves **{}**, o valores das suas **chaves (campos) : informação** e caso tenha mais de um campo, deve separa-los por vírgula ( **,** )

```
let nomeObjeto = { Campo: informação, Campo: informação, Campo: informação };
```

Conseguimos comparar um **OBJETO** a uma carteira de motorista, que nada mais é do que campos que possuem as informações.

```
1  let documento = {
2    nome: 'Daniel',
3    sobrenome: 'Porto',
4    rg: '22.333.444-5',
5    cpf: '111.111.111-11',
6    idade: 29,
7    time: 'Fluminense',
8    brasileiro : true
9  }
```

Vamos criar nosso primeiro **OBJETO** e ao passar o cursor ele já nos mostra os tipos de dados de cada campo que criamos.

```
1  const documentoIdentidade = {
2    nome: 'Daniel',
3    sobrenome: 'Porto',
4    cpf: '111.111.111-11',
5    empresa: 'Hashtag',
6    idade: 29,
7  };
8                log(...data: any[]): void
9  console.log()
```

```
1  const documentoIdentidade = {
2    nome: 'Dan
3    sobrenome:
4    cpf: '111.
5    empresa: '
6    idade: 29,
7  };
8
9  console.log(documentoIdentidade);
```

```
const documentoIdentidade: {
    nome: string;
    sobrenome: string;
    cpf: string;
    empresa: string;
    idade: number;
}
```

Verificando o retorno desse objeto, repare que nosso terminal retorna toda a estrutura do objeto.

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_2\aula12> node .\objeto.js
{
  nome: 'Daniel',
  sobrenome: 'Porto',
  cpf: '111.111.111-11',
  empresa: 'Hashtag',
  idade: 29
}
```

# Módulo 2 – Object em Javascript ( 4 / 5)

Além de manipularmos o objeto por inteiro, nós também conseguimos acessar os campos individualmente, ou seja, é possível acessar cada informação de cada campo separadamente.

Para acessarmos as informação, escrevemos o nosso do objeto + ponto( . ) + nome da chave.

O próprio VS Code já te retorna a lista de campos que existe no seu objeto.

```
10  console.log(documentoIdentidade);
11
```

```
cpf                    (property) cpf: string
empresa
idade
nome
sobrenome
```

Acessando seus campos individualmente, verificamos o seu retorno:

```
1   const documentoIdentidade = {
2     nome: 'Daniel',
3     sobrenome: 'Porto',
4     cpf: '111.111.111-11',
5     empresa: 'Hashtag',
6     idade: 29,
7   };
8
9   // console.log(documentoIdentidade);
10  console.log(documentoIdentidade.nome);
11  console.log(documentoIdentidade.idade);
```

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_2\aula12> node .\objeto.js
Daniel
29
```

# Módulo 2 – Object em Javascript ( 5 / 5 )

Conseguimos acessar as informações de cada campo individualmente também, utilizando a sintaxe nomeObjeto[ "nomeChave" ].

Dentro de um objeto temos **pares de chave : valor**, também chamado como "dicionário".

```
1   const documentoIdentidade = {
2     nome: 'Daniel',
3     sobrenome: 'Porto',
4     cpf: '111.111.111-11',
5     empresa: 'Hashtag',
6     idade: 29,
7   };
8
9   // console.log(documentoIdentidade);
10  console.log(documentoIdentidade.nome);
11  console.log(documentoIdentidade.idade);
12  console.log(documentoIdentidade['cpf']);
```

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_2\aula12> node .\objeto.js
Daniel
29
111.111.111-11
```

# Módulo 2 – Array em Javascript ( 1 / 4 )

Lembra quando dissemos que uma variável é como uma caixinha com uma etiqueta, na qual você guarda uma informação?

Pense no **Array** como uma "super caixa", cheia de compartimentos para preenchermos com dados diferentes! Outra forma de olhar para isso é entendendo que temos uma **"LISTA"** de valores.

**ARRAY**

**Array** são objetos semelhantes as listas, pois são usados para armazenar vários valores em uma única variável, ou seja, são estruturas que guardam mais de um valor assim como o Objeto.

O tamanho do Array e nem os tipos de elementos são fixos, ou seja, posso declarar um array com tamanhos diferentes e tipos de dados diferentes.

A sintaxe para declarar que uma variável é um Array é utilizar na sua atribuição de valores os colchetes [].

```
1   const investimentoMensal = [];
```

Para popular um **Array** deve ser passado as informações entre vírgulas ( , ).

```
1   const investimentoMensal = [1000, 2000, 3000, 2000, 1500, 1200, 1300];
```

Ao declararmos um array e atribuir valores a ele a sua tipagem pode alterar entre **any**(qualquer tipo de dado), para um array específico, como nosso exemplo de um **Array de number**.

```
JS array.js > ...   const investimentoMensal: number[]
1   const investimentoMensal = [1000, 2000, 3000, 2000, 1500, 1200, 1300];
```

Ele poderia ser um tipo de Array de Strings, Booleanos e  além de tipos específicos, um **Array aceita diferentes tipos de informações a cada elemento**.

```
array.js > ...   const arrayDeStrings: (string | number)[]
1   // con
2   const arrayDeStrings = ['string1', 2];
```

# Módulo 2 – Array em Javascript ( 4 / 4 )

Para acessarmos os valores únicos do Array, precisamos entender que a cada elemento de um array ele possuirá um **índice.**

Os índices sempre iniciam no valor 0, ou seja, a posição 1 do meu array terá índice 0, a posição 2 terá índice 1, e assim por diante, independentemente do valor armazenado.

```
1   const investimentoMensal = [1000, 2000, 3000, 2000, 1500, 1200, 1300];
2   // const arrayDeStrings = ['string1', 2];
3
4   // console.log(investimentoMensal);
5   console.log(`O valor investido no mês de janeiro é: ${investimentoMensal[0]}`);
6   console.log(
7     `O valor investido no mês de fevereiro é: ${investimentoMensal[1]}`
8   );
9   console.log(`O valor investido no mês de março é: ${investimentoMensal[2]}`);
10  console.log(`O valor investido no mês de abril é: ${investimentoMensal[3]}`);
11  console.log(`O valor investido no mês de maio é: ${investimentoMensal[4]}`);
12  console.log(`O valor investido no mês de junho é: ${investimentoMensal[5]}`);
13
```

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_2\aula13> node ./array.js
O valor investido no mês de janeiro é: 1000
O valor investido no mês de fevereiro é: 2000
O valor investido no mês de março é: 3000
O valor investido no mês de abril é: 2000
O valor investido no mês de maio é: 1500
O valor investido no mês de junho é: 1200
```

Acabamos de ver sobre **Objetos** e **Arrays** duas estruturas que aceitam mais valores dentro de sua variável e que são do tipo Referência, nesse momento iremos aplicar alguns conceitos sobre as nomenclaturas de seus elementos.

Existe uma outra nomenclatura para o campo / chave de um objeto, quando declarmos um objeto no Vs Code e passamos o cursor em cima dele, reparamos que a palavra **property** aparece. Isso indica que um campo do objeto é referenciado como **propriedade do objeto**.

```
let   (property) nome: string
    nome: 'Let it Be',
    artista: 'Beatles'
}
```

As posições de cada elemento de um Array é chamado de índice.

```
7   let listaDeBandas = ['Beatles', 'Rolling Stones', 'Led Zeppelin', 'Cássia Eller', 'Chico Buarque'];
8
9   console.log(listaDeBandas[4])
```

# Módulo 2 – Nomenclatura e Mutabilidade ( 2 / 4 )

Ao tentarmos acessar um índice de um elemento do array que não existe, o Javascript é flexível e nos retorna o valor padrão para quando uma variável não é declarada, undefined.

```
7   let listaDeBandas = [
8     'Beatles',
9     'Rolling Stones',
10    'Led Zeppelin',
11    'Cássia Eller',
12    'Chico Buarque',
13  ];
14
15  console.log(listaDeBandas[4]);
16  console.log(listaDeBandas[5]);
```

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_2\aula14> node ./tiposReferencia.js
Chico Buarque
undefined
```

Pela flexibilidade que o Javascript possui, podemos por exemplo, adicionar novos elementos ao nosso Array da seguinte maneira:

```
7   let listaDeBandas = [
8     'Beatles',
9     'Rolling Stones',
10    'Led Zeppelin',
11    'Cássia Eller',
12    'Chico Buarque',
13  ];
14
15  console.log(listaDeBandas[4]);
16  listaDeBandas[5] = 'Skank';
17  console.log(listaDeBandas[5]);
```

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_2\aula14> node ./tiposReferencia.js
Chico Buarque
Skank
```

Com isso, conseguimos adicionar novos elementos, mas também alterar informações dos nosso elementos, utilizando a sintaxe:

**NomeArray[índice da alteração] = atribuição nova;**

```
7   let listaDeBandas = [
8     'Beatles',
9     'Rolling Stones',
10    'Led Zeppelin',
11    'Cássia Eller',
12    'Chico Buarque',
13  ];
14
15  console.log(listaDeBandas[4]);
16  listaDeBandas[5] = 'Skank';
17  listaDeBandas[3] = 'Deep Purple';
18  console.log(listaDeBandas[5]);
19  console.log(listaDeBandas);
```

```
[
  'Beatles',
  'Rolling Stones',
  'Led Zeppelin',
  'Deep Purple',
  'Chico Buarque',
  'Skank'
]
```

Essas alterações também são possíveis de fazer quando falamos de Objetos, o que não é comum em outras linguagens, ou seja, criar novos campos e alterar campos.

```
1   let musica = {
2     nome: 'Let it Be',
3     artista: 'Beatles',
4     album: 'Let it Be',
5   };
```

```
21   musica.album = 'Past Masters';
22   musica.ano = 1969;
23   console.log(musica);
```

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_2\aula14> node ./tiposReferencia.js
{
  nome: 'Let it Be',
  artista: 'Beatles',
  album: 'Past Masters',
  ano: 1969
}
```

Essas alterações que podemos fazer no array e objetos mostram a sua **Mutabilidade**, a maneira como conseguimos manipular essas duas estruturas.

# Módulo 2 – Por que "Tipos Referência" ( 1 / 2 )

Diferente dos **"Tipos de valor"**, cujo tamanho máximo é conhecido, os limites dos tipos referência não são bem definidos. Isso porque o programador pode construir dados tão grandes quanto a sua necessidade (dentro dos limites da máquina que opera esses dados)!
Ou seja, os **"Tipos de valor"** ocupam um espaço X de bits e bytes que caracterizam o seu tamanho e ele já é conhecido.

Já a lógica do **"Tipos Referência**" não existe um valor pré-definido para seu tamanho, ou seja, a nossa lógica da caixinha não sobrevive se eu apenas especificar (de forma bem vaga) que ela deve abrigar um animal.
Um cachorro e um elefante são animais , mas concordam que o tamanho deles é muito diferente?

Então a primeiro momento não conseguimos verificar qual o tamanho que um **"Tipos Referência"** pode ter, apenas quando ele já estiver criado.

# Módulo 2 – Por que "Tipos Referência" ( 2 / 2 )

Sendo assim, a nossa estratégia passa a ser outra! Vamos guardar o dado em um espaço muito maior, e colocar a varável apenas o "endereço desse lugar".

Na nossa analogia um elefante não cabe dentro da caixa, então fazemos o seguinte, vamos colocar ele em um container e dentro da nossa caixinha colocaremos a chave desse container, ou seja, eles são do Tipo referência porque  o que temos na variável é a informação da onde está o nosso dado maior.

A variável não armazena a informação e sim o endereço da informação.

Sabemos que uma variável armazena um dado do Tipo Valor, pois são dados menores ou que possuem seu tamanho definido, já o armazenamento de um Tipo Referência é feita através do endereço daquele dado, por ser um dado de tamanho não definido ou dados muito grandes, nossa variável irá armazenar a referência do local que nossa informação se encontra.

Agora vamos ver algumas particularidades do **Tipo Referência**.

Imagine o cenário que você possui uma carteira e você possui 0,10 centavos, você concorda que esse valor cabe dentro da sua carteira correto? Agora imagine que você possui 10 milhões de reais, muito provável que esse valor não cabe na sua carteira, para você guardar teria que ir até o banco e depositar dentro de uma conta e passamos a carregar um cartão (chave) para acessar esse valor.

Porém a sua carteira foi clonada, dentro do cenário que você tem 0,10 centavos está tudo bem, mas imagine o outro cenário, no qual foi também clonado o seu cartão, ou seja, outra pessoa também possui a chave para sua conta.

# Módulo 2 – As consequências de manipulação de referências ( 2 / 6 )

Para o cenário descrito temos o seguinte código:

Declaramos um objeto que carregará os dados do meu cartão.

Em seguida, declaro uma variável que será uma cópia desse meu objeto.

Ao analisarmos o retorno, verificamos que eles possuem a mesma informação.

```
const cartao1 = {
  nome: 'Daniel Porto',
  saldo: 10000000,
};

const cartao2 = cartao1;

console.log(cartao1);
console.log(cartao2);
```

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_2\aula16> node .\referenceTypeModified.js
{ nome: 'Daniel Porto', saldo: 10000000 }
{ nome: 'Daniel Porto', saldo: 10000000 }
```

Agora, vamos fazer uma alteração no saldo do nosso cartão2.

Nessa linha de código, estamos dizendo que o valor do saldo irá ter seu valor subtraído por um valor.

```
cartao2.saldo -= 2000000;
```

```
cartao2.saldo = cartao2.saldo - 2000000
```

# Módulo 2 – As consequências de manipulação de referências ( 3 / 6 )

Agora vamos observar o que aconteceu ao implementarmos o nosso código.

```
1   const cartao1 = {
2     nome: 'Daniel Porto',
3     saldo: 10000000,
4   };
5
6   const cartao2 = cartao1;
7
8   console.log(cartao1);
9   console.log(cartao2);
10
11  cartao2.saldo -= 2000000;
12
13  console.log(cartao1);
14  console.log(cartao2);
```

Repare que a alteração foi feita na nossa variável cartao2, mas essa alteração foi aplicada em ambos os objetos, em ambas as variáveis.

Isso ocorre porque quando fizemos uma cópia do objeto cartao1, passamos também a referência de onde está armazenada a sua informação.

Utilizando da nossa analogia, é como o cartão clonado que consegue acessar a sua conta e realizar transações.

Esse tipo de manipulação ocorre em todos os Tipo Referência, inclusive no Array.

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_2\aula16> node .\referenceTypeModified.js
{ nome: 'Daniel Porto', saldo: 10000000 }
{ nome: 'Daniel Porto', saldo: 10000000 }
{ nome: 'Daniel Porto', saldo: 8000000 }
{ nome: 'Daniel Porto', saldo: 8000000 }
```

# Módulo 2 – As consequências de manipulação de referências ( 4 / 6 )

Observando o nosso código podemos verificar uma outra particularidade dos Tipo Referências.

Repare que nossos objetos foram declarado como variáveis constantes, mas que nós conseguimos alterar o valor de um dos campos.

Porque de fato o que nossa variável cartao2 está armazenando é o local da nossa informação.

Então as informações do objeto podem ser alteradas, pois o local de onde estão as informações não está sendo alterado, você mantém a referência do lugar.

```
1   const cartao1 = {
2     nome: 'Daniel Porto',
3     saldo: 10000000,
4   };
5
6   const cartao2 = cartao1;
7
8   console.log(cartao1);
9   console.log(cartao2);
10
11  cartao2.saldo -= 2000000;
12
13  console.log(cartao1);
14  console.log(cartao2);
```

```
cartao2 = {
  nome: 'João Lira',
  saldo: 100000000,
};
```

```
C:\Users\danie\JavaScript Impressionador - arquivos\modulo_2\aula16\referenceTypeModified.js:16
cartao2 = {
        ^

TypeError: Assignment to constant variable.
    at Object.<anonymous> (C:\Users\danie\JavaScript Impressionador - arquivos\modulo_2\aula16\referenceTypeModified.js:16:9)
    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)
```

Agora se tentarmos mudar a nossa variável cartao2 e criar um novo objeto dentro dela, ocasionará um erro, pois estamos alterando não apenas as informações, mas sim a referência de onde ela ficará armazenada.

Ou seja, estou fazendo uma nova atribuição a uma variável do tipo constante, e sabemos que isso não é permitido.

Além de tentar armazenar um outro endereço desse novo objeto, porém o cartao2 já possui um endereço de referência.

Você consegue alterar todas as informações do objeto que já existe dentro da variável.

# Módulo 2 – As consequências de manipulação de referências ( 6 / 6 )

Conseguimos aplicar essas peculiaridades também nos Arrays, já que são do Tipo Referência.

E funcionará da mesma forma, conseguimos alterar valores de dentro do Array, mas não podemos atribuir um novo endereço ou informações a ele.

```
const listaAlunos = ['Daniel', 'Lira', 'Alon'];
listaAlunos[3] = 'Renan';

console.log(listaAlunos);
```

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_2\aula16> node .\referenceTypeModified.js
{ nome: 'Daniel Porto', saldo: 10000000 }
{ nome: 'Daniel Porto', saldo: 10000000 }
{ nome: 'Daniel Porto', saldo: 8000000 }
{ nome: 'Daniel Porto', saldo: 8000000 }
[ 'Daniel', 'Lira', 'Alon', 'Renan' ]
```

```
const listaAlunos = ['Daniel', 'Lira', 'Alon'];
listaAlunos = ['Renan', 'Hecio'];

console.log(listaAlunos);
```

```
C:\Users\danie\JavaScript Impressionador - arquivos\modulo_2\aula16\referenceTypeModified.js:17
listaAlunos = ['Renan', 'Hecio'];
            ^

TypeError: Assignment to constant variable.
    at Object.<anonymous> (C:\Users\danie\JavaScript Impressionador - arquivos\modulo_2\aula16\referenceTypeModified.js:17:13)
    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)
```

Módulo 3

# FUNÇÕES

FUNÇÕES

FUNÇÕES

```js
aula3_1.js
console.log('Olá, boa noite!');
console.log('Pegue aqui a sua comanda!');
console.log('Bom apetite!!');
console.log('---------------------------');
console.log('Olá, boa noite!');
console.log('Pegue aqui a sua comanda!');
console.log('Bom apetite!!');
console.log('---------------------------');
console.log('Olá, boa noite!');
console.log('Pegue aqui a sua comanda!');
console.log('Bom apetite!!');
console.log('---------------------------');
console.log('Olá, boa noite!');
console.log('Pegue aqui a sua comanda!');
console.log('Bom apetite!!');
console.log('---------------------------');
```

```
---------------------------
Olá, boa noite!
Pegue aqui a sua comanda!
Bom apetite!!
---------------------------
Olá, boa noite!
Pegue aqui a sua comanda!
Bom apetite!!
---------------------------
Olá, boa noite!
Pegue aqui a sua comanda!
Bom apetite!!
---------------------------
Olá, boa noite!
Pegue aqui a sua comanda!
Bom apetite!!
---------------------------
```

Vamos analisar o exemplo ao lado, onde temos uma sequência de comandos de console.log() e o resultado do seu retorno.

Imagine um **cenário de restaurante**, onde cada pessoa que entrar será abordado por um **computador** com as seguintes **instruções**:

**"Olá, boa noite! Pegue aqui a sua comanda! Bom apetite!!".**

Na situação ao lado, 4 pessoas entraram nesse restaurante, consequentemente 4 instruções foram escritas e retornadas.

Mas imagine se 10, 20 ou 100 pessoas entrassem no restaurante e tivéssemos que escrever manualmente todos esses comando de console.log() para instruir o computador.

Pense no tempo gasto para sua execução, possíveis erros de digitação que poderiam gerar se o número de pessoas aumentasse e o tamanho que o nosso programa teria **REPETINDO** a mesma informação.

O programa **Funcionaria**, correto? Mas seria **a forma mais inteligente** de se realizar essa tarefa??

Não seria melhor utilizar apenas um bloco de instruções de console.log(), ou seja, encapsular essas informações para que elas sejam apenas uma coisa, no nosso exemplo um Processo de pegar a comanda.

aula3_1.js **Bloco de instrução Pegar Comanda**

```
console.log('Olá, boa noite!');
console.log('Pegue aqui a sua comanda!');
console.log('Bom apetite!!');
console.log('---------------------------');
```

Esse conceito existe em JavaScript e é chamado de **FUNÇÃO**. Uma Função é conjunto de instruções que executa uma tarefa pré-definida. Para usar uma **função**, você deve **defini-la** em algum lugar do seu programa e depois **chamá-la / referenciá-la** para executar ela.

A sintaxe da **definição ou declaração de uma função** é utilizar a palavra chave **function**, seguida por **nome** da função, **parênteses ()** e as **instruções entre chaves { }**.

Palavra reservada

Nome da função

Bloco de instruções

Escopo da função (local das instruções)

Referência para executar a função

aula3_1.js > ...

```
function pegarComanda() {
  console.log('Olá, boa noite!');
  console.log('Pegue aqui a sua comanda!');
  console.log('Bom apetite!!');
  console.log('---------------------------');
}

pegarComanda();
```

Para executarmos a nossa função precisamos chamá-la ou referenciá-la, a sintaxe é **nome da função** + **parenteses()**.

```js
aula3_1.js > ...
function pegarComanda() {
  console.log('Olá, boa noite!');
  console.log('Pegue aqui a sua comanda!');
  console.log('Bom apetite!!');
  console.log('--------------------------');
}

pegarComanda();
```

Dessa forma ao executarmos novamente o programa, teremos o seu retorno:

```
PS C:\Users\danie\JavaScript Impressionador\modulo_3\aula1> node ./aula3_1.js
Olá, boa noite!
Pegue aqui a sua comanda!
Bom apetite!!
--------------------------
```

Agora para aplicarmos o exemplo do restaurante fica muito mais simples, limpo e legível. Para cada pessoa que entrar, podemos chamar nossa função novamente que o seu retorno será o bloco de instruções.

```js
aula3_1.js > ...
function pegarComanda() {
  console.log('Olá, boa noite!');
  console.log('Pegue aqui a sua comanda!');
  console.log('Bom apetite!!');
  console.log('--------------------------');
}

pegarComanda();
pegarComanda();
pegarComanda();
pegarComanda();
```

```
--------------------------
Olá, boa noite!
Pegue aqui a sua comanda!
Bom apetite!!
--------------------------
Olá, boa noite!
Pegue aqui a sua comanda!
Bom apetite!!
--------------------------
--------------------------
Olá, boa noite!
Pegue aqui a sua comanda!
Bom apetite!!
--------------------------
Olá, boa noite!
Pegue aqui a sua comanda!
Bom apetite!!
--------------------------
```

```js
aula3_2.js > ...
let numeroDaVez = 1;
console.log(`O número da vez é o ${numeroDaVez}`);
console.log(
  `O número da vez (${numeroDaVez}) ao quadrado vale ${numeroDaVez ** 2}`
);
numeroDaVez = 2;
console.log(`O número da vez é o ${numeroDaVez}`);
console.log(
  `O número da vez (${numeroDaVez}) ao quadrado vale ${numeroDaVez ** 2}`
);
```

```
PS C:\Users\danie\JavaScript Impressionador\modulo_3\aula3_2> node ./aula3_2.js
O número da vez é o 1
O número da vez (1) ao quadrado vale 1
O número da vez é o 2
O número da vez (2) ao quadrado vale 4
PS C:\Users\danie\JavaScript Impressionador\modulo_3\aula3_2>
```

Acima temos com um **programa** da seguinte maneira:

Declaramos a **variável numeroDaVez** e atribuímos um valor inicial de **1**. Imprimimos o conteúdo dentro do **console.log()**, sendo o valor inicial e outro elevado a potência de 2. Alteramos o valor da **variável numeroDaVez por 2** e repetimos o comando **console.log()** e seu conteúdo. Ao executarmos o programa, vimos que ele funciona.

Mas sabemos que repetir linhas de códigos com os mesmos conteúdos, não é a maneira mais eficiente ou inteligente de se escrever um programa, **correto?**

Imaginemos o cenário onde queremos realizar esse programa com mais números ( 3, 4 ...), como vimos anteriormente, a **repetição de linhas de códigos iguais são desnecessárias**, pois causam grande problemas na leitura do programa e **tempo gasto para sua manutenção**, mesmo tendo sua funcionalidade correta, o programa continuaria retornando as nossas instruções mas de maneira **ineficaz**.

Então precisamos aplicar o **conceito de função**, ou seja, precisamos **declarar uma função** que seja capaz de realizar a tarefa de "pegar" um número e imprimir as instruções com os textos.

aula3_2.js > ...

```js
let numeroDaVez = 1;
console.log(`O número da vez é o ${numeroDaVez}`);
console.log(
  `O número da vez (${numeroDaVez}) ao quadrado vale ${numeroDaVez ** 2}`
);
numeroDaVez = 2;
console.log(`O número da vez é o ${numeroDaVez}`);
console.log(
  `O número da vez (${numeroDaVez}) ao quadrado vale ${numeroDaVez ** 2}`
);
numeroDaVez = 3;
console.log(`O número da vez é o ${numeroDaVez}`);
console.log(
  `O número da vez (${numeroDaVez}) ao quadrado vale ${numeroDaVez ** 2}`
);
numeroDaVez = 4;
console.log(`O número da vez é o ${numeroDaVez}`);
console.log(
  `O número da vez (${numeroDaVez}) ao quadrado vale ${numeroDaVez ** 2}`
);
```

```js
aula3_2.js
let numeroDaVez = 1;

function calcularQuadrado() {
  console.log(`O número da vez é o ${numeroDaVez}`);
  console.log(
    `O número da vez (${numeroDaVez}) ao quadrado vale ${numeroDaVez ** 2}`
  );
}
```

Lembrando que uma função para ser declarada é necessário utilizar a palavra reservada **function nomeDaFuncao ( ) { bloco de instruções }** e depois ser chamada para sua execução **nomeDaFuncao()**.

No nosso exemplo, analisamos que os comandos de **console.log()** estavam sendo repetidos e a variável **numeroDaVez** estava sendo alterada / atualizada.

Você concorda que as linhas que estavam sendo repetidas, serão o nosso bloco de instruções, que ficam dentro da nossa função?

Mas você deve estar se perguntando, o que fazemos com a variável, pois se declararmos ela dentro da função, iremos ter o mesmo problema, só que nesse caso precisaríamos criar diversas funções iguais com variáveis diferentes? **ERRADO!**

A nossa função precisa ser **autossuficiente**, ou seja, ela precisa poder captar diversos **numeroDaVez** e aplicar a tarefa definida.

Se observarmos dentro da nossa função, estamos utilizando a **String Template**, ou seja, a nossa função precisa capturar essa variável de algum lugar.

```
JS aula3_2.js > calcularQuadrado
1  function calcularQuadrado() {
2    console.log(`O número da vez é o ${numeroDaVez} );
3    console.log(
4      `O número da vez (${numeroDaVez}) ao quadrado vale ${numeroDaVez ** 2}`
5    );
6  }
7
8  calcularQuadrado();
9
```

```
C:\Users\danie\JavaScript Impressionador\modulo_3\aula3_2\aula3_2.js:2
  console.log(`O número da vez é o ${numeroDaVez}`);
                                   ^

ReferenceError: numeroDaVez is not defined
    at calcularQuadrado (C:\Users\danie\JavaScript Impressionador\modulo_3\aula3_2\aula3_2.js:2:38)
    at Object.<anonymous> (C:\Users\danie\JavaScript Impressionador\modulo_3\aula3_2\aula3_2.js:8:1)
    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
```

E ao tentar executar nossa função dessa maneira iremos receber o **erro** que a **variável numeroDaVez não está sendo definida** em nenhum lugar do nosso código.

Para resolvermos o erro da nossa função, precisamos entender mais um detalhe da declaração da mesma, os **Parênteses ()**. Entendemos que as **chaves {} são a delimitação das nossas instruções**, mas para que serve os parênteses () ??

Quando a sua função precisa de uma informação para ser executada, ela será definida dentro dos parênteses(). Essa informação é chamada de **PARÂMETRO**, ou seja, é a variável declarada na definição de uma função:

```
JS aula3_2.js > calcularQuadrado        parâmetro
1  function calcularQuadrado(numeroDaVez) {
2    console.log(`O número da vez é o ${numeroDaVez}`);
3    console.log(
4      `O número da vez (${numeroDaVez}) ao quadrado vale ${numeroDaVez ** 2}`
5    );
6  }
```

Então para usar essa função precisamos passar uma informação para ela:

```
JS aula3_2.js > ...
1  function calcularQuadrado(numeroDaVez) {
2    console.log(`O número da vez é o ${numeroDaVez}`);
3    console.log(
4      `O número da vez (${numeroDaVez}) ao quadrado vale ${numeroDaVez ** 2}`
5    );
6  }
7
8  calcularQuadrado(1);
```

```
PS C:\Users\danie\JavaScript Impressionador\modulo_3\aula3_2> node ./aula3_2.js
O número da vez é o 1
O número da vez (1) ao quadrado vale 1
```

Dessa forma, transformamos nossa função e ela se torna autossuficiente, ou seja, através do Parâmetro, podemos executar nossa funções com informações diferentes.

```js
aula3_2.js > ...
function calcularQuadrado(numeroDaVez) {
  console.log(`O número da vez é o ${numeroDaVez}`);
  console.log(
    `O número da vez (${numeroDaVez}) ao quadrado vale ${numeroDaVez ** 2}`
  );
}

calcularQuadrado(1);
calcularQuadrado(2);
calcularQuadrado(3);
calcularQuadrado(4);
calcularQuadrado(5);
calcularQuadrado(6);
calcularQuadrado(7);
```

```
PS C:\Users\danie\JavaScript Impressionador\modulo_3\aula3_2> node ./aula3_2.js
O número da vez é o 1
O número da vez (1) ao quadrado vale 1
O número da vez é o 2
O número da vez (2) ao quadrado vale 4
O número da vez é o 3
O número da vez (3) ao quadrado vale 9
O número da vez é o 4
O número da vez (4) ao quadrado vale 16
O número da vez é o 5
O número da vez (5) ao quadrado vale 25
O número da vez é o 6
O número da vez (6) ao quadrado vale 36
O número da vez é o 7
O número da vez (7) ao quadrado vale 49
```

Mas existe uma **diferença de nomenclatura** quando estamos definindo e chamando a função. Vimos que um Parâmetro é a variável que declaramos junto com a criação da função, a informação necessária.

Agora quando referenciamos a função, ou seja, chamamos ela para ser executada. A informação que escrevemos dentro do parênteses () é chamado de **ARGUMENTO**, ou seja, é o valor atribuído que a nossa função precisa para funcionar.

```
calcularQuadrado(1);
```

**Argumento**

# Módulo 3 – Funções - Retorno (Return) (1 / 5)

Hoje vamos falar um pouco sobre matemática, mas de uma forma computacional, você se lembra das equações de 2° grau, aquelas equações que são $ax^2 + bx + c = 0$, em que a, b e c são números reais, conhecidos como coeficientes da equação.

Uma das famosas equações de 2°grau é a fórmula de Bhaskara, que é um método resolutivo para equações de 2°grau utilizado para encontrar raízes a partir de coeficientes da equação:

$$x1, x2 = \frac{-b +- \sqrt{(b^2 - 4ac)}}{2a}$$

Fórmula de Bhaskara

E para conseguirmos esses valores, precisamos da equação que mostre quais são os coeficientes:

$$f(x) = ax^2 + bx + c \longrightarrow f: y = 2x^2 + 3x - 2$$

Na matemática, substituiríamos os números, e conseguiríamos achar uma parábola, onde y = 0 e x descobriríamos.

Mas calma! Que a ideia é apenas entender a teoria, e não aplicar todas essas fórmulas matemáticas. Vamos aprender a como programar e fazer o computador realizar essa tarefa para nós.

Começamos criando uma função, que terá como tarefa resolver as nossas equações com fórmulas Bhaskaras.

```
JS aula3_3.js > resolverBhaskara
1   function resolverBhaskara() {}
```

Lembrando que declaramos a função, com a palavra chave **function**, nomeamos a nossa função, entre os **parênteses ()** colocaremos os **parâmetros** (caso precisemos para executar nossa função) e entre **chaves {}** nossas **instruções.**

Uma boa prática para declararmos funções, é ela ter **um objetivo específico**, ou seja, é muito importante uma função realizar uma tarefa e não várias. A nossa função **resolverBhaskara**, tem como objetivo resolver a equação, mas precisamos de algumas informações para facilitar a criação da mesma. Por exemplo, não seria interessante criar uma função que calcule a raiz quadrada e utilizar o resultado dentro da nossa função **resolverBhaskara**.

Esse forma de criar funções menores e mais objetiva, **facilita a manutenção** do nosso código futuramente e caso eu precise realizar mais alguma tarefa que envolva calcular a raiz quadrada, conseguiríamos **reutilizar** a função, facilitando o processo.

Vamos criar a **função calcularRaizQuadrada**, ela receberá um **parâmetro** que será o número que queremos "retirar" a raiz quadrada (lembrando que utilizamos o parâmetro porque precisamos de uma variável para executar a nossa função) e nomearemos ela como **"base"**, o nome de uma parâmetro pode ser qualquer um, mas vale da boa prática nomearmos com um nome que faça sentido ao que a variável representa.

```
function calcularRaizQuadrada(base) {
    base**(1/2);
    console.log(base**(1/2));
}
```

```
function calcularRaizQuadrada(base) {
    base**(1/2);
    console.log(base**(1/2));
}
```

Mas se repararmos na nossa função, apenas damos a tarefa de imprimir o valor da raiz quadrada, não estamos utilizando esse valor, pois não há uma entrega do valor, ou seja, o comando console.log() apenas imprimi na tela, mas não me permite atribuir o valor que a função está devolvendo para uma variável.

Então como "entregamos" esse valor para atribuir à uma variável?

```
let valorRaizQuadrada = calcularRaizQuadrada(4);
```

Para que isso aconteça, precisamos utilizar uma palavra reservada chamada **RETURN** dentro da nossa função.
O **Return** é uma declaração que **finaliza a execução** de uma **função**, ou seja, a função é parada e ela **retorna valores especificados** para onde a função for chamada.

Colocamos o **return** ao final da nossa função e pedimos que ela **retorne ("entregue")** o **valor especificado**, neste caso, a raiz quadrada.

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

O **return** faz com que a **função** tenha me entregue um resultado que eu posso **utilizar ou armazenar**, mas sabendo que ao utilizar o **return** geramos um **valor**.

Mas vimos anteriormente, que a declaração do **return** faz com que a nossa **função finalize** logo após. Então ao tentarmos adicionar novas tarefas a nossa função, ela não conseguirá executar, pois a chamada da função já foi finalizada.

```
JS aula3_3.js > ...
1   function resolverBhaskara() {}
2
3   function calcularRaizQuadrada(base) {
4     return base ** (1 / 2);
5
6     console.log('Eu queria escrever isso aqui na tela. Será que eu consigo?');
7   }
8
9   let valorRaizQuadrada = calcularRaizQuadrada(4);
10
```

**Resultado ao executar o programa**

```
PS C:\Users\danie\JavaScript Impressionador\modulo_3\aula3_3> node ./aula3_3.js
PS C:\Users\danie\JavaScript Impressionador\modulo_3\aula3_3>
```

Repare que o **VS Code** deixa o comando console.log() com uma cor mais apagada. E ao passarmos o cursor em cima dele, verificamos que a mensagem **"Unreachable code detected"**, essa mensagem indica que é um código que não será executado, nesse caso é porque ele está após o **return (última execução da função)**.

Para executar o comando console.log() precisamos colocá-lo antes do return.

```
JS aula3_3.js > calcularRaizQuadrada
1   function resolverBhaskara() {}
2
3   fu
4     Unreachable code detected. ts(7027)
5     Quick Fix... (Ctrl+.)
6     console.log('Eu queria escrever isso aqui na tela. Será que eu consigo?');
7   }
8
9   let valorRaizQuadrada = calcularRaizQuadrada(4);
```

## 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);
```

```
JS aula3_3.js > resolverBhaskara
1  function resolverBhaskara() {
2      (-b +- (b^2 - 4ac) ^1/2)/2a
3  }
4
5  function calcularRaizQuadrada(base) {
6    return base ** (1 / 2);
7  }
```

Fórmula matemática de Bhaskara

Vamos revisar alguns conceitos para continuarmos.

Vimos que se a **função** necessita de uma informação especifica para ser executada, passamos ela dentro do parênteses e é chamado de **Parâmetro**. Quando chamamos / executamos a função passamos o valor como **Argumento.**

O **return** é utilizado no final e tem o objetivo de **retornar o resultado** da função, para utilizarmos, e **encerrar** a nossa função.

Incrível! Agora que relembramos esses conceitos, continuaremos a nossa função **resolverBhaskara**.

No primeiro momento, iremos escrever de forma matemática a **fórmula de Bhaskara** dentro da nossa função, para **interpretar** e escrevermos em **JavaScript**.

# Módulo 3 – Funções - Múltiplos parâmetros e ordem de execução (2 / 5)

Quando começamos a interpretar a fórmula matemática para a linguagem JavaScript, temos o seguinte resultado:

Repare que possuímos uma **função que calcula a raiz quadrada** para nós, então invocamos ela e atribuímos o **argumento**, ou seja, o **valor** que queremos calcular a raiz.

```
JS aula3_3.js > resolverBhaskara
1  function resolverBhaskara() {
2      (-b +- (b^2 - 4ac) ^1/2)/2a
3      const x1 = (-b + calcularRaizQuadrada(b^2 - 4ac))/2a;
4      const x2 = (-b - calcularRaizQuadrada(b^2 - 4ac))/2a;
5  }
6
7  function calcularRaizQuadrada(base) {
8    return base ** (1 / 2);
9  }
```

**Fórmula em Javascript**

Um ponto importante parar analisarmos também, é de onde está vindo os **valores a, b** e **c**? Note que estamos utilizando mas não atribuímos o seu valor.

Se nossa **função** precisa dessas informações para ser executada, passaremos os **valores de a, b** e **c** como seus **parâmetros**.

```
JS aula3_3.js > ...
1  function resolverBhaskara(a, b, c) {
2      const x1 = (-b + calcularRaizQuadrada(b^2 - 4ac))/2a;
3      const x2 = (-b - calcularRaizQuadrada(b^2 - 4ac))/2a;
4  }
5
6  function calcularRaizQuadrada(base) {
7    return base ** (1 / 2);
8  }
```

Mais uma alteração é necessária no nosso código, para que o Javascript interprete a fórmula dentro da linguagem e não na fórmula matemática, ou seja, o JavaScript não entende o que é **b^2 – 4ac e 2a**. Para isso utilizaremos os **operadores aritméticos**.

```
const x1 = (-b + calcularRaizQuadrada(delta)) / (2 * a);
const x2 = (-b - calcularRaizQuadrada(delta)) / (2 * a);
```

Agora sim! Perceba até que o erro que o VS Code apontava, para esses valores, parou.

Observando nosso código, sabemos que o Javascript sabe interpretar, e ele resolverá primeiro o que está no parênteses da função auxiliar **calcularRaizQuadrada**, encontrará o resultado, para depois calcular a raiz desse resultado e retornará o valor para a fórmula que está dentro da função resolverBhaskara.

Mas acredito que podemos melhorar o nosso programa. Vamos atribuir a equação para uma variável e utilizar essa variável como parâmetro da função **calcularRaizQuadrada**. Transformando em um código mais simples e legível.

```
1  function resolverBhaskara(a, b, c) {
2    const delta = b ** 2 - 4 * a * c;
3    const x1 = (-b + calcularRaizQuadrada(delta) / (2 * a));
4    const x2 = (-b - calcularRaizQuadrada(delta) / (2 * a));
5  }
6
7  function calcularRaizQuadrada(base) {
8    return base ** (1 / 2);
9  }
```

# Módulo 3 – Funções - Múltiplos parâmetros e ordem de execução (4 / 5)

Um detalhe importante sobre o conceito **função**, é que podemos **criar variáveis** dentro dela, mas essa variável pode ser acessada em outro lugar fora dela. Nesse momento você não precisa entender muito bem sobre os acessos das variáveis, é apenas um spoiler das aulas futuras.

Nossa função está pronta, iremos adicionar um console.log() para verificarmos se a execução está correta e chamar nossa função para ser executada:

```
PS C:\Users\danie\JavaScript Impressionador\modulo_3\aula3_3> node ./aula3_3.js
Os dois valores de X dos pontos da parábola no eixo X são 0.5 e -2
```

```javascript
function resolverBhaskara(a, b, c) {
  const delta = b ** 2 - 4 * a * c;
  const x1 = (-b + calcularRaizQuadrada(delta)) / (2 * a);
  const x2 = (-b - calcularRaizQuadrada(delta)) / (2 * a);

  console.log(
    `Os dois valores de X dos pontos da parábola no eixo X são ${x1} e ${x2}`
  );
}

function calcularRaizQuadrada(base) {
  return base ** (1 / 2);
}

resolverBhaskara(2, 3, -2);
```

# Módulo 3 – Funções - Múltiplos parâmetros e ordem de execução (5 / 5)

Para nossa análise final, destacaremos algumas informações sobre nosso código.

```
function resolverBhaskara(a, b, c) {
  const delta = b ** 2 - 4 * a * c;
  const x1 = (-b + calcularRaizQuadrada(delta)) / (2 * a);
  const x2 = (-b - calcularRaizQuadrada(delta)) / (2 * a);

  console.log(
    `Os dois valores de X dos pontos da parábola no eixo X são ${x1} e ${x2}`
  );
}

function calcularRaizQuadrada(base) {
  return base ** (1 / 2);
}

resolverBhaskara(2, 3, -2);
```

Primeiro, repare que utilizamos na função **resolverBhaskara 3 parâmetros**.

O número de parâmetros será de acordo com as informações que a função precisa para executar, e passaremos cada uma delas ( **sem parâmetros ()**, com **um ou mais parâmetros (param1, param2, param3, ...)**.

O segundo ponto é que criamos a **função calcularRaizQuadrada** depois de chamá-la/ usá-la dentro da **função resolverBhaskara**, e falamos que o JavaScript é lido de cima para baixo, então como ele resolveu isso? Como o nosso programa executou sem gerar nenhum erro?

A leitura e interpretação é um pouco mais complicado. Através das palavras reservadas **function, return, ponto e vírgula, regras de escrita do JavaScript**, o **motor do JavaScript** consegue identificar as tarefas que estamos tentando realizar em cada parte do nosso código.

É como se o motor fizesse a leitura prévia de tudo, por exemplo, leu todas as funções e armazenou essas informações, quando ele for executar elas, ele já tem esses valores armazenados, então ele conhece essas funções. E dessa forma ao chegar na chamada de **calcularRaizQuadrada**, ele já conhece o seu funcionamento e já consegue dar o retorno.

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 está 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**.

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**!

aula3_5.js > ...

Escopo Global

```
1  const primeiraVariavelDoCodigo = 'texto inicial';
2
3  function printToConsole() {
4    console.log(primeiraVariavelDoCodigo);
5  }
6
7  printToConsole();
```

Escopo Local

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 função
4     const primeiraVariavelDoCodigo = 'segundo texto';
5     console.log(primeiraVariavelDoCodigo);
6   }
7
8   printToConsole();
```

```
PS C:\Users\danie\JavaScript Impressionador\modulo_3\aula3_5> node .\aula3_5.js
segundo texto
```

Ou seja, se entendermos que o **conceito de Escopo** nos permite, em determinadas partes do código, ter acesso a determinadas variáveis, a questão levantada sobre o redeclararmos a variável **primeiraVariavelDoCodigo** poder ter seu valor alterado se torna **verdadeiro dentro do Escopo local da função printToConsole**, sem apresentar nenhum erro.

Mas se tentarmos alterar novamente o valor da variável, só que nesse momento dentro do escopo da função, o mesmo erro irá ocorrer, pois não podemos reatribuir valores para variáveis do tipo const que estão localizadas no mesmo escopo.

```
JS aula3_5.js > printToConsole > primeiraVariavelDoCodigo
1  const primeiraVariavelDoCodigo = 'texto inicial';
2
3  function printToConsole() {
4      const primeiraVariavelDoCodigo = 'segundo texto';
5      const primeiraVariavelDoCodigo = 'terceiro texto';
6      console.log(primeiraVariavelDoCodigo);
7  }
8
9  printToConsole();
```

Para continuarmos com alguns exemplos precisamos saber que é possível declarar funções dentro de outras funções. Ela é declarada com a sintaxe de uma função só que dentro de outra função.

Esse tipo de declaração chama-se Função aninhada e inicialmente é acessível apenas ao Escopo que a contém, mas sua referência pode ser retornada para outro escopo (Mas não se preocupe nesse momento em entender a importância desse contexto. Exploraremos mais sobre isso ao longo do curso).

```
JS aula3_5.js > printToConsole        Escopo Global
1   const primeiraVariavelDoCodigo = 'texto inicial';
2
3   function printToConsole() {   Escopo Local função PrintToConsole
4     const primeiraVariavelDoCodigo = 'segundo texto';
5     function secondFunction() {Escopo Local função secondFunction
6       console.log('imprimindo segunda mensagem');
7     }
8     console.log(primeiraVariavelDoCodigo);
9     secondFunction();
10  }
11
12  printToConsole();
13
```

```
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 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. (/home/milene/javascript_hashtag/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 secondFuction** 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 3 – Funções - Escopos (11 / 11)

Agora que conseguimos identificar os tipos de Escopos 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.**

# Módulo 3 – O que são métodos (1 / 7)

Vamos recapitular alguns conceitos que vimos no módulo anterior e nesse. Lembra-se que aprendemos sobre os Tipo Referência, e um deles era o **Objeto**.

O **Objeto** irá agrupar informações com contextos semelhantes, então iremos começar nosso exemplo criando um objeto CASA e armazenando algumas informações desse contexto.

```
let casa = {
  cor: 'white',
  sala: 15,
  cozinha: 10,
  banheiro: 8,
  quarto: 20,
};
```

Ao lado definimos um objeto casa, e repare que dentro do nosso objeto adicionamos os **campos** / chaves e seus respectivos **valores**. Sabemos que o nosso objeto pode **armazenar valores de todos os tipos**, como Strings e Numbers.
Nosso objeto irá guardar a cor da nossa casa e a metragem ($m^2$) de cada cômodo.

Para continuarmos, vamos criar uma **Função** que retorno o tamanho total que a casa possui.

```
function calcularTamanhoTotalCasaEmM2(
  tamanhoSala,
  tamanhoCozinha,
  tamanhoBanheiro,
  tamanhoQuarto
) {
  const tamanhoTotal =
    tamanhoSala + tamanhoQuarto + tamanhoCozinha + tamanhoBanheiro;
  console.log(`O tamanho da casa é de ${tamanhoTotal}m²`);
  return tamanhoTotal;
}
```

```
calcularTamanhoTotalCasaEmM2(
  casa.sala,
  casa.cozinha,
  casa.banheiro,
  casa.quarto
);
```

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_3\aula_7> node .\method.js
O tamanho da casa é de 53m²
```

Definimos a nossa **Função** e ela receberá 4 **parâmetros**, ou seja, para ser executada ela precisa de 4 informações, que serão os tamanhos do cômodos.

Em seguida definimos uma **variável** para armazenar a soma dos tamanhos de cada cômodo, iremos printar na tela utilizando a **String Template**.

Para deixar nossa função completa vamos **retornar** o **tamanhoTotal**, para caso precisemos utilizar em outro momento do nosso código.

Para executarmos nossa função, precisamos referenciá-la e adicionar os 4 **argumentos** que ela precisa, ou seja, agora vamos preencher as informações para que nossa função consiga se executar.

Os argumentos serão os valores que estão armazenados dentro do nosso Objeto, para acessar basta chamar o nome do **objeto.nomeDoCampo.**

Podemos verificar o seu retorno executando o programa.

# Módulo 3 – O que são métodos (3 / 7)

Lembra que a **Função**, também é definida como um **Tipo Referência**, e que como falamos no início que o nosso **Objeto** ele irá guardar informações de contexto semelhante e que ele aceitas todos os tipos dentro dele?

Você concorda que a nossa Função, está retornando o valor do Tamanho total da casa e que ele se encaixa dentro do contexto do nosso objeto? Afinal, ele está armazenando características dessa casa?

Agora observe como conseguimos manipular o nosso objeto:

Em primeiro lugar, vamos criar um novo campo dentro do nosso objeto, com o nome da função que acabamos de criar, dentro desse campo será armazenado uma função, então iremos declarar ela no próprio objeto.

Repare que o nome da função será a propriedade do objeto **calcularTamanhoTotalCasaEmM²**.

```
let casa = {
  cor: 'white',
  sala: 15,
  cozinha: 10,
  banheiro: 8,
  quarto: 20,
  calcularTamanhoTotalCasaEmM2: function() {
    |
  }
};
```

Nosso próximo passo é adicionar o **bloco de instrução** que definimos dentro da nossa função para ela printar e retornar o tamanho total, mas repare que nesse momento os parâmetros / argumentos que precisamos da nossa função, está dentro do escopo do Objeto, no qual nossa função também está vinculada.

```
let casa = {
  cor: 'white',
  sala: 15,
  cozinha: 10,
  banheiro: 8,
  quarto: 20,
  calcularTamanhoTotalCasaEmM2: function () {
    const tamanhoTotal =
      tamanhoSala + tamanhoQuarto + tamanhoCozinha + tamanhoBanheiro;
    console.log(`O tamanho da casa é de ${tamanhoTotal}m²`);
    return tamanhoTotal;
  },
};
```

```
let casa = {
  cor: 'white',
  sala: 15,
  cozinha: 10,
  banheiro: 8,
  quarto: 20,
  calcularTamanhoTotalCasaEmM2: function () {
    const tamanhoTotal = casa.sala + casa.cozinha + casa.banheiro + casa.quarto;
    console.log(`O tamanho da casa é de ${tamanhoTotal}m²`);
    return tamanhoTotal;
  },
};
```

Como não queremos mais as informações vindo de um lugar externo do local que minha função está, conseguimos acessar esses valores de forma mais prática e dinâmica. Chamando os valores diretamente do Objeto que estamos.

# Módulo 3 – O que são métodos (5 / 7)

Uma curiosidade que conseguimos aplicar dentro do nosso Objeto é o **THIS**.

**This** não é uma variável, é uma palavra-chave que refere-se a um objeto que está executando o trecho de código atual, ou seja, ele faz referência ao objeto que está executando a função atual.

Você não pode alterar o valor de **this**.

Conseguimos utilizar o **this** no caso abaixo, como referência ao objeto que está sendo chamado dentro da nossa função.

```
calcularTamanhoTotalCasaEmM2: function () {
  const tamanhoTotal = this.sala + this.cozinha + this.banheiro + this.quarto;
  console.log(`O tamanho da casa é de ${tamanhoTotal}m²`);
  return tamanhoTotal;
```

Agora ficou muito mais simples de executarmos a nossa função, iremos chamar nosso objeto e o campo que está a nossa função armazenada, da seguinte maneira:

```
1   let casa = {
2     cor: 'white',
3     sala: 15,
4     cozinha: 10,
5     banheiro: 8,
6     quarto: 20,
7     calcularTamanhoTotalCasaEmM2: function () {
8       const tamanhoTotal = this.sala + this.cozinha + this.banheiro + this.quarto;
9       console.log(`O tamanho da casa é de ${tamanhoTotal}m²`);
10      return tamanhoTotal;
11    },
12  };
13
14  casa.calcularTamanhoTotalCasaEmM2();
```

Repare que o retorno da nosso programa será o mesmo que anteriormente, porém conseguimos criar um código muito mais simples.

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_3\aula_7> node .\method.js
O tamanho da casa é de 53m²
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_3\aula_7> node .\method.js
O tamanho da casa é de 53m²
```

# Módulo 3 – O que são métodos (7 / 7)

Quando uma função está ligada a um objeto, o nome que damos é **MÉTODO**.

**MÉTODO**

Um método JavaScript é um objeto contendo a definição function() e são armazenados como propriedades do objeto, ou seja, quando uma propriedade armazena uma função, ela se torna o que chamamos de método. Ele pode retornar algo ou não.

Existem métodos já existentes dentro do Javascript, mas também conseguimos criar nossos próprios métodos.

Um exemplo que viemos praticando muito, sem você saber que estava utilizando um método é o nosso **CONSOLE.LOG()**.

**Log()** é uma função que está ligada ao **objeto Console** (objeto que aponta para o ambiente que você está executando o Javascript / terminal), ou seja, o **Log() é um método** que é responsável por capturar a informação e printar no terminal.

Não é apenas o tipo Objeto que possui métodos. Em Javascript, vamos entender que os métodos estão ligados a vários tipos e cada um deles possui métodos específicos para serem utilizados.

Vamos criar uma variável que armazene uma informação do tipo **Number**, quando chamamos a nossa variável e passamos um **ponto ( . )** após ela, o VS Code nos mostra uma lista de possíveis métodos que podemos utilizar com aquele tipo.

Repare que o método é representado no VS Code por um cubo roxo, e ao passarmos o cursor em cima do primeiro, ele nos mostra que é um ( método ) Number.toExponential.

Nesse momento vamos conceituar e explicar um pouco desses métodos que aparecem na listagem do VS Code.

**MÉTODOS TIPO NUMBER**

**toExponential :**
- O método **toExponential** () converte um **número** em uma notação exponencial.

**toLocaleString** :
- O **toLocaleString** () retorna um **número** como uma string, usando o formato de idioma local. O formato do idioma depende da configuração de localidade em seu computador.

**toPrecision** :
- O método **toPrecision**() em Javascript é usado para formatar um número com uma precisão ou comprimento específico.

**valueOf :**
- O método **valueOf**() retorna o valor primitivo contido no objeto **Number**.

**toFixed** :

- O método **toFixed** () converte um **número** em uma string. O método **toFixed** () arredonda a string para um **número** especificado de casas decimais.
- Esse método recebe como parâmetro quantas casas decimais você quer converter um número.
- Antes de aplicar o método ele realizar uma conversão para String para conseguir executar o método.

```
let a = 2.00000001;

console.log(a);
console.log(a.toFixed(3));
```

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_3\aula_8> node .\typeMethods.js
2.00000001
2.000
```

**toString :**

- O **toString** () é um método **Number** usado para converter um **número** em uma representação de string do **número** . O VS Code identifica o Number com a cor amarela e a String com a cor branca.

```
let a = 2.00000001;

console.log(a);
console.log(a.toFixed(3));
console.log(a.toString());
```

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_3\aula_8> node .\typeMethods.js
2.00000001
2.000
2.00000001
```

Existem vários métodos que são associados ao tipo String e vamos conhecer alguns deles nesse momento.

**MÉTODOS TIPO STRING**

**charAt:**

- O método **charAt** () retorna o caractere no índice especificado em uma **string** . O índice do primeiro caractere é 0, o segundo caractere é 1 e assim por diante.
- Precisamos passar como parâmetro o valor da posição do caractere que queremos retornar.

```
let b = 'texto da aula de metodos';

console.log(b.charAt(2));
```

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_3\aula_8> node .\typeMethods.js
x
```

**concat:**

- O método **concat** () junta duas ou mais **strings** . O método **concat** () não altera as **strings** existentes . O método **concat** () retorna uma nova **string** .

```
let b = 'texto da aula de metodos';

console.log(b.concat(' que eu já dominei totalmente'));
```

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_3\aula_8> node .\typeMethods.js
texto da aula de metodos que eu já dominei totalmente
```

**split:**

- O **split()**método pega um padrão e divide a String em uma lista ordenada de substrings procurando pelo padrão, coloca essas substrings em uma matriz e retorna a matriz, ou seja, ele retorna um **array com as substrings.**

Se passar o cursor em cima do nome do método, o VS Code mostra o que esse método faz.

```
JS typeMethods.js > [@] b
1   // let a = 2.0
2
3   // console.log
4   // console.log
5   // console.log
6
7   let b = 'texto
8
9   console.log(b.split);
```

```
(method) String.split(separator: string | RegExp, limit?: number | undefined): string[] (+1 overload)
```

Split a string into substrings using the specified separator and return them as an array.

*@param* `separator` — A string that identifies character or characters to use in separating the string. If omitted, a single-element array containing the entire string is returned.

*@param* `limit` — A value used to limit the number of elements returned in the array.

```
let b = 'texto da aula de métodos';

console.log(b.split('da'));
```

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_3\aula_8> node .\typeMethods.js
[ 'texto ', ' aula de métodos' ]
```

**split:**

- O parâmetro do método, será o local, palavra ou caractere que queremos procurar para fazer a divisão. Por exemplo, no caso do espaço, se utilizarmos ele no método, conseguimos retornar um array composto pelas palavras que estão presentes na frase.

```
let b = 'texto da aula de métodos';

console.log(b.split(' '));
```

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_3\aula_8> node .\typeMethods.js
[ 'texto', 'da', 'aula', 'de', 'métodos' ]
```

Os métodos também podem estar associado aos **tipos de dados**, e não apenas aos dados / informações que estamos utilizando dentro de uma variável. O tipo **NUMBER** possui suas propriedades e métodos específicos.

**Nan:**

- A propriedade **Number.NaN** representa Not-A-Number (Não-Número). Equivalente a NaN.

**parseInt:**

- O **método Number.parseInt()** converte um argumento de *string* e retorna um inteiro da raiz ou base específica.

```
let c = '10110';
console.log(Number.parseInt(c));
```

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_3\aula_8> node .\typeMethods.js
10110
```

# Módulo 3 – Métodos associados a tipos (7 / 8)

Conseguimos utilizar métodos com o resultado de outro método. Da seguinte maneira:

Utilizaremos um exemplo com **método parseInt** e com o **método isNan** (que verifica se o número é do tipo NaN ou não).

Ao tentarmos transformar um conteúdo de uma string, que não é um número, verificamos que seu retorno será do tipo NaN.

```
let b = 'texto da aula de métodos';

let c = '10110';
console.log(Number.parseInt(c));
console.log(Number.parseInt());
```

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_3\aula_8> node .\typeMethods.js
10110
NaN
```

E podemos utilizar outro método para verificar o retorno do método parseInt. E veremos um valor booleano para seu retorno.

```
let b = 'texto da aula de métodos';

let c = '10110';
console.log(Number.parseInt(c));
console.log(Number.isNaN(Number.parseInt(b)));
```

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_3\aula_8> node .\typeMethods.js
10110
true
```

# Módulo 3 – Métodos associados a tipos (8 / 8)

Não se preocupe em decorar os métodos, afinal eles são muitos. Alguns são utilizados com uma frequência muito alta, assim como tem outros métodos que são utilizados em momentos muito específicos. Mas ao longo de sua carreira como programador, você irá sempre pesquisar por eles.

Então novamente, não se preocupe em decorar os métodos, mas sim compreender o que são e porque são utilizados.

Uma curiosidade que o VS Code também nos retorna, é que além dos métodos serem identificados pelo cubo roxo, podemos identificar as propriedades que aquele tipo possui e elas são identificadas por um retângulo azul.

**length:**

- É uma propriedade que retorna o tamanho da string, ou seja, o comprimento de uma string é referido como o número total de caracteres que ela contém.

```
let b = 'texto da aula de métodos';

console.log(b.length);
```

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_3\aula_8> node .\typeMethods.js
24
```

# Módulo 3 – Métodos de Arrays (1 / 3)

Existem muitos métodos disponíveis para manipular e acessar elementos em um Array em JavaScript. Aqui vamos ver 3 de uso comum no seu dia a dia. Lembrando que existem muito outro métodos, e você irá aprender e conhecer ao longo de sua jornada como desenvolvedor.

```
const listaNotas = [
  { nome: 'Daniel', nota: 05 },
  { nome: 'Pedro', nota: 15 },
  { nome: 'Lucas', nota: 12 },
  { nome: 'Fabricio', nota: 16 },
  { nome: 'Izabelle', nota: 18 },
  { nome: 'Nathalia', nota: 20 },
  { nome: 'Amanda', nota: 20 },
  { nome: 'João', nota: 20 },
  { nome: 'Marcus', nota: 12 },
  { nome: 'Sergio', nota: 13 },
  { nome: 'Diego', nota: 14 },
  { nome: 'Renan', nota: 11 },
  { nome: 'Rodrigo', nota: 09 },
  { nome: 'Messi', nota: 00 },
  { nome: 'CR7', nota: 00 },
];
```

Aqui vamos iniciar com um cenário, onde temos um array com objetos dentro dele, e cada objeto armazena o campo de Nome e Nota e seus respectivos valores.

Vamos aprender através desse exemplo os métodos que não modificam o array original:

**find:**

- O método **find()** retorna o **valor** do **primeiro elemento** do array que satisfaz a função de teste fornecida. Caso contrário, undefinedé retornado.

**filter:**

- O método **filter()** cria um novo array com todos os elementos que passaram no teste implementado pela função ficaram.

**map:**

- O método **map()** invoca a função callback passada por argumento para cada elemento do Array e devolve um novo Array como resultado.

**filter:**

- O método **filter()** é responsável por gerar um novo Array resultado de um filtro estabelecido pelo programador.
- Ele receberá uma funçõ como parâmetro com uma regra estabelecida e irá implementar em cada elemento do array, verificando se o booleano retornado será true ou false, e toda vez que o elemento for true, ele armazenará essa informação em um novo array.

```
//filter - Método responsável por gerar um novo array resultado de um filtro estabelecido pelo programador
const alunosAprovados = listaNotas.filter(function (aluno) {
  return aluno.nota >= 12;
});

console.log(alunosAprovados);
```

```
{ nome: 'Fabricio', nota: 16 },
{ nome: 'Izabelle', nota: 18 },
{ nome: 'Nathalia', nota: 20 },
{ nome: 'Amanda', nota: 20 },
{ nome: 'João', nota: 20 },
{ nome: 'Marcus', nota: 12 },
{ nome: 'Sergio', nota: 13 },
{ nome: 'Diego', nota: 14 }
```

**find:**

- O método **find()** diferente do filter encontra apenas o primeiro elemento que corresponde a regra criada não retornará todos os elementos que sigam essa regra.

```
const alunosQueTiraramZero = listaNotas.find(function (aluno) {
  return aluno.nota === 0;
});

console.log(alunosQueTiraramZero);
```

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_3\aula_9> node .\methodsInArrays.js
{ nome: 'Messi', nota: 0 }
```

**map:**

- O método **map()** pega uma lista e retorna uma nova lista, com o mesmo tamanho, respeitando uma regra ou condição criada para mapear esse array, armazenando as informações que estamos buscando.

```
const listaNomesAlunosAprovados = alunosAprovados.map(function (aluno) {
  return aluno.nome;
});

console.log(listaNomesAlunosAprovados);
```

```
[
  'Pedro',     'Lucas',
  'Fabricio',  'Izabelle',
  'Nathalia',  'Amanda',
  'João',      'Marcus',
  'Sergio',    'Diego'
]
```

Módulo 4

# 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 nosso 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.

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 está 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 > ...
function calculateSquareRoot(radicand) {
  return radicand ** (1 / 2);
}
```

**IGUAL**

```js
function calcularRaizQuadrada(base) {
  return base ** (1 / 2);
}
```

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

```javascript
function calculateSquareRoot(radicand) {
  return radicand ** (1 / 2);
}

function calculatePythagoras(side1, side2) {
  const sum = side1 ** 2 + side2 ** 2;
  return calculateSquareRoot(sum);
}
```

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(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(`O tamanho da hipotenusa de um triangulo 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
JS calculatePythagoras.js > ...
function calculateSquareRoot(radicand) {
  return radicand ** (1 / 2);
}

function calculatePythagoras(side1, side2) {
  const sum = side1 ** 2 + side2 ** 2;
  return 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: 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.

# Módulo 4 – Depuração e Desenvolvimento Eficiente – Console log: uma primeira abordagem (1 / 10)

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.

# Módulo 4 – Depuração e Desenvolvimento Eficiente – Console log: uma primeira abordagem (2 / 10)

Começando com o seguinte cenário:

Na função **calculateSquareRoot** ao invés de elevarmos nossa **variável radicand** a **potência ( ** )** de ½ , multiplicarmos **( * )** o **radicand** por ½.

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)}`
);
```

```
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
```

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: NaN
```

Repare 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**.

# Módulo 4 – Depuração e Desenvolvimento Eficiente – Console log: uma primeira abordagem (3 / 10)

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.

Analisando o nosso código, identificamos que realmente tentamos usar um **texto 'sum'** como **argumento** para a **função calculateSquareRoo**t 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?

# Módulo 4 – Depuração e Desenvolvimento Eficiente – Console log: uma primeira abordagem (4 / 10)

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 utiliza-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.

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.

# Módulo 4 – Depuração e Desenvolvimento Eficiente – Console log: uma primeira abordagem (5 / 10)

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
O 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 indicio 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.

# Módulo 4 – Depuração e Desenvolvimento Eficiente – Console log: uma primeira abordagem (6 / 10)

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.

```
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');
}
```

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.**

# Módulo 4 – Depuração e Desenvolvimento Eficiente – Console log: uma primeira abordagem (7 / 10)

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 conseguir encontrar o local do erro.

```
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
```

```
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)}`
);
```

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".

```
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
```

**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.

```
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)}`
);
```

# Módulo 4 – Depuração e Desenvolvimento Eficiente – Console log: uma primeira abordagem (9 / 10)

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.

```
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 construto de qualquer linguagem de programação. Eles existem em todas elas, a gente tem vários nomes para eles: bugs, erros, exceções, etc.

```
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!

```
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)}`
);
```

# Módulo 4 – Depuração e Desenvolvimento Eficiente – Console log: uma primeira abordagem (10 / 10)

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
```

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.

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 especifico 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 entende-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.

```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>Pythagoras' Web Page</title>
    <link rel="stylesheet" href="style.css" />
  </head>
  <body>
    <h2>Calcule a Hipotenusa de qualquer Triângulo Retângulo</h2>
    <div class="answer-container">
      <img
        src="assets/pythagoras.jpeg"
        alt="Busto de Pitágoras."
        id="pythagoras"
      />
      <div id="answer" class="hidden-element"></div>
    </div>
    <div class="inputs-container">
      <label for="side-1">Primeiro Cateto</label>
      <input id="side-1" type="text" />
      <label for="side-2">Segundo Cateto</label>
      <input id="side-2" type="text" />
      <button id="calculate-hypotenuse">Calcular Hipotenusa</button>
    </div>
    <script src="/calculatePythagoras.js"></script>
  </body>
</html>
```

CSS

JAVASCRIPT

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.hmtl**, 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.

Repare que existem **duas caixas de textos** ( 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 */** .

calculatePythagoras.js > ...

```js
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)}`
// );

const firstSideRef = document.getElementById('side-1');
const secondSideRef = document.getElementById('side-2');
const calculateButton = document.getElementById('calculate-hypotenuse');
const answerDiv = document.getElementById('answer');

function onCalculateButtonClick() {
  const firstSide = firstSideRef.value;
  const secondSide = secondSideRef.value;
  // console.log('function executed');
  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>`;
}
```

Ao comentar / retirar os console.log() do nosso código e executar o programa novamente, nada irá ser retornado no DevTools – Console.

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.

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.

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**.

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.

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.

```
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.

Continuando a depuração do nosso código, isso quer dizer, executando as demais linhas de código com o nosso debugger. Chegamos a seguinte situação:

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 entende-la completamente, o que precisa compreender que existem um campo dentro dessa variável que retorna o nosso texto e retorna o resultado.

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 }
```

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.

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 vam 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 a1+a2 = cateto A e b1+b2 = 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.

```
<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 Hipotenus
</div>
```

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');
```

A primeira modificação no nosso código é adicionar as referências dos novos elementos que criamos no arquivo HTML.

Em seguida, implementaremos  a lógica de somar os lados de A e B, para retornarem o valor de cada lado do cateto corretamente.

```
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:

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 recorde 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.

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.

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 especifico 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 utiliza-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 declara 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.

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    SQL CONSOLE: MESSAGES

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
```

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',
  'C:\\Users\\danie\\JavaScript Impressionador\\modulo_4\\aula4_5\\calculatePythagoras.js',
  '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 ignora-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' ]
A hipotenusa de um triangulo de lados 3 e 4 é: 5
```

SUBARRAY

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.

```
   0     1     2     3
[ '2', '3', '1', '2' ]
```

Para finalizar nosso código iremos alterar nosso retorne 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**.

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.

Ao iniciar a execução do **Run and Debug**, o controle de fluxo estará disponível na tela para você executar as próximas instruções.

# Módulo 4 – Depuração e Desenvolvimento Eficiente – DevTools no VS Code (11 /15)

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.

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
```

Dessa forma nosso programa será executado via terminal vinculado com o Debugger , para isso a informação Debugger attached, será impressa no terminal.

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:

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.

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 5

# 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 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.

**ESTRUTURAS CONDICIONAIS**

Servem para estabelecer blocos de código que apenas devem ser executados caso uma condição seja estabelecida, ou seja a estrutura condiciona 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 retorna verdadeiro ou falso, e o algoritmo executará o bloco de comandos relativos a este resultado.

**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.

# Módulo 5 – Blocos Condicionais - IF (3/5)

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.

# Módulo 5 – Blocos Condicionais - IF (4 /5)

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.

```js
const usuario = { nome: 'Daniel', idade: 29, time: 'Fluminense' };
const mensagemDeBoasVindas1 = 'Bem vindo, torcedor Tricolor!';
const mensagemDeBoasVindas2 = 'Bem vindo, torcedor Rubro-Negro!';

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

const mensagemDeEscolha =
  'Clique no setor para o qual deseja comprar seu ingresso!';
const mensagemFinal = 'Divirta-se no Maracanã!!!';

console.log(mensagemDeEscolha);
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!!!
```

```js
const usuario = { nome: 'Daniel', idade: 29, time: 'Vasco' };
const mensagemDeBoasVindas1 = 'Bem vindo, torcedor Tricolor!';
const mensagemDeBoasVindas2 = 'Bem vindo, torcedor Rubro-Negro!'

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

const mensagemDeEscolha =
  'Clique no setor para o qual deseja comprar seu ingresso!';
const mensagemFinal = 'Divirta-se no Maracanã!!!';

console.log(mensagemDeEscolha);
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!!!
```

# Módulo 5 – Blocos Condicionais - IF (5 /5)

```
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);
18
```

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'.

# Módulo 5 – IF / ELSE IF / ELSE (1 /7)

**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.**

# Módulo 5 – IF / ELSE IF / ELSE (2 /7)

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

aula5_3 > JS exampleCode.js > ...

```js
const usuario = { nome: 'Daniel', idade: 29, time: 'Fluminense' };
const mensagemDeBoasVindas1 = 'Bem vindo, torcedor Tricolor!';
const mensagemDeBoasVindas2 = 'Bem vindo, torcedor Rubro-Negro!';

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

const mensagemDeEscolha =
  'Clique no setor para o qual deseja comprar seu ingresso!';
const mensagemFinal = 'Divirta-se no Maracanã!!!';

console.log(mensagemDeEscolha);
console.log(mensagemFinal);
```

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!!!
```

```js
const usuario = { nome: 'Daniel', idade: 29, time: 'Flamengo' };
const mensagemDeBoasVindas1 = 'Bem vindo, torcedor Tricolor!';
const mensagemDeBoasVindas2 = 'Bem vindo, torcedor Rubro-Negro!';

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

const mensagemDeEscolha =
  'Clique no setor para o qual deseja comprar seu ingresso!';
const mensagemFinal = 'Divirta-se no Maracanã!!!';

console.log(mensagemDeEscolha);
console.log(mensagemFinal);
```

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 especifica 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.

# Módulo 5 – IF / ELSE IF / ELSE (3 /7)

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 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!!!
```

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.

# Módulo 5 – IF / ELSE IF / ELSE (4 /7)

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, imprimiríamos 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 if .....else**, ou seja, "SE...Caso contrário  se .....Caso 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
}
```

# Módulo 5 – IF / ELSE IF / ELSE (5 /7)

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 cado 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!!!
```

# Módulo 5 – IF / ELSE IF / ELSE (6 /7)

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.

# Módulo 5 – IF / ELSE IF / ELSE (7 /7)

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.

# Módulo 5 – SWITCH CASE (1 /7)

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.

# Módulo 5 – SWITCH CASE (2 /7)

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 pararmos 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**.

# Módulo 5 – SWITCH CASE (3 /7)

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 estuturar 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.

# Módulo 5 – SWITCH CASE (4 /7)

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:
41      console.log(mensagemDeBoasVindasGeral);
42  }   break;
```

**OPCIONAL**

# Módulo 5 – SWITCH CASE (5 /7)

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!!!
```

# Módulo 5 – SWITCH CASE (6 /7)

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 comtemplada.

```
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:

# Módulo 5 – OPERADOR TERNÁRIO (1 /6)

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**

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 tambḿe optar pelo operador ternário para representar o cenário

# Módulo 5 – OPERADOR TERNÁRIO (3 /6)

**IF / ELSE**

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

**OPERADOR TERNÁRIO**

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

No código acima mudaremos a Estrutura simples do IF / ELSE em um Operador Concidional 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).

# Módulo 5 – OPERADOR TERNÁRIO (4 /6)

**IF / ELSE IF / ELSE**

```javascript
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**

```javascript
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 Concidional Ternário.

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

# Módulo 5 – OPERADOR TERNÁRIO (5 /6)

```
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.

# Módulo 5 – OPERADOR TERNÁRIO (6 /6)

**Condições Booleans** 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 BOOLENA**

```
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.

# Módulo 5 – DO... WHILE (1 /5)

**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 booelana**, que retornará true ou false para o código saber se ele realizará a repetição ou não.

# Módulo 5 – DO... WHILE (2 /5)

Queremos que o nosso Portal do Torcedor continue sendo executado enquanto os torcedores estiverem utilizando ele, ou seja, um torcedor atrás do outro deve conseguir utilizar o Portal, e esse Portal precisa se reinicializar quando outro torcedor vier comprar os ingressos, e também precisamos criar uma condição para que ele Pare de funcionar quando não tiver mais torcedores para comprar o ingresso, ou os ingressos acabaram, ou o horário de venda já acabou.

Para isso vamos utilizar a estrutura de repetição **DO..WHILE**, primeiro vamos observar a estrutura que o nosso código está.

```
1   const usuario = { nome: 'Daniel', idade: 17, time: 'Sport' };
2   const mensagemDeBoasVindas1 = 'Bem vindo, torcedor Tricolor!';
3   const mensagemDeBoasVindas2 = 'Bem vindo, torcedor Rubro-Negro!';
4   const mensagemDeBoasVindas3 = 'Bem vindo, torcedor Vascaíno!';
5   const mensagemDeBoasVindasGeral = 'Bem vindo, amante do esporte!';
6   const mensagemDeEscolha =
7     'Clique no setor para o qual deseja comprar seu ingresso!';
8   const mensagemFinal = 'Divirta-se no Maracanã!!!';
9
10  usuario.time === 'Fluminense' && usuario.idade >= 18
11    ? console.log(mensagemDeBoasVindas1)
12    : usuario.time === 'Flamengo' && usuario.idade >= 18
13    ? console.log(mensagemDeBoasVindas2)
14    : usuario.idade >= 18
15    ? console.log(mensagemDeBoasVindasGeral)
16    : console.log('Não vendemos ingresso para torcedores menores de 18 anos');
17
18
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   SQL CONSOLE: MESSAGES   Filte
C:\Program Files\nodejs\node.exe .\aula5_5\exampleCode.js
Não vendemos ingresso para torcedores menores de 18 anos
Clique no setor para o qual deseja comprar seu ingresso!
Divirta-se no Maracanã!!!
```

Continua sendo o código que construímos nesse módulo, apenas posicionamos todas **as declarações das nossas variáveis** no início para uma melhor organização de código.

Já deixamos o **objeto de um usuário**, mas na prática o torcedor iria fornecer os dados dele, mas nesse momento para executarmos nossa problema/solução, iremos utilizar um usuário fixo.

O nosso **bloco de estrutura de condicional** irá ser aplicado para todos os torcedores. Pois são as condições que temos para saber qual a mensagem a aparecer para o torcedor.

# Módulo 5 – DO... WHILE (3 /5)

O nosso código deverá possuir um variável que armazene a quantidade de repetições / de iterações que o nosso laço de repetição fará, porque ele será um contador que controla quantas vezes o bloco de instrução foi executado. E se cada pessoa executar o nosso programa 1 vez, queremos que essa variável armazene, todas as vezes que for executado até chegar a 50.000 vezes. E ai o programa será encerrado.

**Variável de controle de iteração**

```
10    let numeroDeRepeticoes = 0;
```

```
10    let numeroDeRepeticoes = 0;  Contador
11
12    do { Faça / Execute pelo menos 1 vez o próximo bloco de instrução
13      usuario.time === 'Fluminense' && usuario.idade >= 18
14        ? console.log(mensagemDeBoasVindas1)
15        : usuario.time === 'Flamengo' && usuario.idade >= 18
16        ? console.log(mensagemDeBoasVindas2)
17        : usuario.idade >= 18
18        ? console.log(mensagemDeBoasVindasGeral)
19        : console.log('Não vendemos ingresso para torcedores menores de 18 anos');
20
21      console.log(mensagemDeEscolha);
22      console.log(mensagemFinal);
23    } while (numeroDeRepeticoes < 50000); Enquanto for verdadeira essa condição
```

Então a nossa estrutura possui uma variável que é um **contador de fluxo**, o bloco de estrutura condiciona ficará dentro do nosso bloco **DO**, pois queremos que ele execute essas instruções. E adicionaremos a condição de PARADA no nosso **WHILE** que será quando atingir 50000 vezes, o programa encerra.

# Módulo 5 – DO... WHILE (4 /5)

Tem um detalhe que está faltando,  o nosso código está armazenando em uma variável quantas vezes a nossa estrutura irá se repetir, mas como eu faço para o meu programa incrementar a cada iteração +1 na minha variável?
Porque a forma que está escrito o nosso código, ela está armazenando 0, o **Do** irá executar uma vez, entraremos na condição do **While ( numeroDeRepeticoes  < 50000)**, **numeroDeRepeticoes tem valor 0**, ou seja, ele reiniciará a estrutura de repetição.

Mas você concorda, que a estrutura do nosso código está dizendo que **sempre o numeroDeRepeticoes é 0**, ou seja, a condição do **WHILE** nunca será falsa, pois 0 é menor 50000, isso era um **loop infinito**. Então o nosso código ficará sendo executado infinitamente, e isso é um problema correto?

Para solucionarmos esse problema, precisamos **alterar o valor da variável de controle** a cada iteração que estamos fazendo em nossa estrutura de repetição.
E iremos utilizar o **Operador de Incremento ++**, ou seja, eu quero aumentar em +1 o número que está armazenada na variável.
Então toda vez que o nosso programa passar pelo bloco de instrução ele irá **incrementar +1 a variável numeroDeRepeticoes**.

```
21    console.log(mensagemDeEscolha);
22    console.log(mensagemFinal);
23    numeroDeRepeticoes++;   +1 a cada iteração
24  } while (numeroDeRepeticoes < 50000);
```

Nosso programa irá executar uma vez, na segunda o valor do contador será 1, na iteração 3, seguindo a iteração 4, e assim por diante, até chegar a 50000 e ele Parar.
Pois na iteração 50001, a condição se torna falsa e o programa sai da estrutura de repetição.

# Módulo 5 – DO... WHILE (5 /5)

Para facilitar o teste do nosso código, iremos delimitar que a nossa estrutura de repetição precise ser executada 5 vezes, e você consegue utilizar o Debug para depurar passo a passo e observar o comportamento da variável numeroDeRepeticoes a cada iteração.

# Módulo 5 – WHILE (1 /9)

A partir dessa aula iremos apresentar um problema único e vamos explorar as diversas formas de resolver esse exemplo, utilizando estruturas de repetição diferentes. Então nós vamos iniciar o nosso programa que envolverá a Sequência de Fibonacci.

A **Sequência de Fibonacci** é um sequência numérica proposta pelo matemático italiano "Leonardo de Pisa" e que desperta fascínio por representar números e padrões identificados em toda a natureza.

A **Sequência de Fibonacci** se dá através da definição do número atual com a soma de seus dois antecessores e ela é infinita.

0,1, **1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584**

**Os coelhos de Fibonacci**

Para exemplificá-la Fibonacci utilizou a reprodução de coelhos idealizados que demoram um mês para amadurecer, produzem todos os meses dois filhotes (um macho e uma fêmea) e nunca morrem,
ou seja , ela representa a quantidade de casais de coelhos mês a mês. Observando com mais cuidado, pode-se perceber que qualquer termo posterior dessa sequência é obtido adicionando os dois termos anteriores.

0,1, **1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584**

O valor 0 é considerado o termo 0 de fibonacci, o primeiro 1 é o termo 1 de fibonacci, e seguindo o valor 1 será o termo 2 e assim por diante...

A sequência de Fibonacci é encontrado em diversos momentos da natureza e por isso é tão famoso esse conceito, repare nas imagens a seguir, o padrão de pétalas das flores pertencem aos termos de Fibonacci

**8 pétalas**

**13 pétalas**

**3 pétalas**

# Módulo 5 – WHILE (4 /9)

Agora vamos criar um programa que gere os números da sequência de fibonacci, mas lembrando como é uma sequência infinita, iremos estipular até qual termo queremos representar no nosso programa.

E para isso vamos utilizar uma estrutura de repetição chama **WHILE**, que é muito parecida com o **Do... While**, porém com algumas características diferentes.

```
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.

Nosso programa terá como funcionalidade retornar a sequência de fibonacci até o termo que escolhermos. Para isso vamos definir um **função e criar duas variáveis** que irão representar os **dois primeiros termos da tabela: 0 e 1.**

```
1  function calculateFibonacciTerm(termNumber) {
2    let lastTerm = 1;
3    let secondToLast = 0;
4  }
```

Após definirmos nossa variável, temos que tratar duas possíveis situação, caso seja passado como Argumento o 0, ou seja, um termo que já sabemos qual é, e também vamos por enquanto aplicar essa lógico no termo 1.

Para isso precisamos criar uma condição que verifique que a **variável termNumber** é 0 ou 1 e o retorno será eles mesmo nessa condição.

```
1  function calculateFibonacciTerm(termNumber) {
2    let lastTerm = 1;
3    let secondToLast = 0;
4
5    if (termNumber === 0) {
6      return 0;
7    }
8    if (termNumber === 1) {
9      return 1;
10   }
11 }
```

# Módulo 5 – WHILE (6 /9)

Definiremos uma variável que irá armazenar o termo que queremos começar a realizar as iterações, como já temos o termo 0 e o termo 1, iremos atribuir o valor 2 para essa variável. E vamos fazer essa definição antes de iniciarmos nossa estrutura de repetição.

```
let currentIteration = 2;
while (){

}
```

```
let currentIteration = 2;
while (currentIteration <= termNumber){

}
```

E vamos adicionar a condição a nossa estrutura de repetição WHILE, que será definida como enquanto o **currentIteration** não for maior ou igual ao **termNumber**, ou seja enquanto não for igual ou maior ao valor do termo que iremos definir como nosso argumento, ele deve realizar o bloco de instrução dentro do **WHILE**.

# Módulo 5 – WHILE (7 /9)

```
let currentIteration = 2;
while (currentIteration <= termNumber) {
  let auxVar = lastTerm + secondToLast;

}
return lastTerm;
```

E antes de definirmos qual bloco de instrução será executado dentro do **While**, iremos finalizar nosso código retornando o **lastTerm**, que irá representar o último termo que calculamos.

Dentro do nosso **While** iremos definir uma variável auxiliar, para armazenar a soma das variáveis **lastTerm** e **secondToLast.**

Atualizamos a variável **secondToLast** com o valor da **lastTerm,** para que na próxima iteração ela já tenha o valor correto da sua próxima posição.

```
let currentIteration = 2;
while (currentIteration <= termNumber) {
  let auxVar = lastTerm + secondToLast;
  secondToLast = lastTerm;

}
return lastTerm;
```

# Módulo 5 – WHILE (8 /9)

Seguindo, atualizamos também a nossa **lastTerm**, que receberá o valor da nossa auxiliar, ou seja, ela armazenou a soma dos termos, e atualizamos sua posição também.

```
let currentIteration = 2;
while (currentIteration <= termNumber) {
  let auxVar = lastTerm + secondToLast;
  secondToLast = lastTerm;
  lastTerm = auxVar;
}
return lastTerm;
```

E realizamos um incremento no nosso **currentIteration**, que atualizará o nosso contador de iterações para ser verificado na nossa condição.

```
let currentIteration = 2;
while (currentIteration <= termNumber) {
  let auxVar = lastTerm + secondToLast;
  secondToLast = lastTerm;
  lastTerm = auxVar;
  currentIteration++;
}
return lastTerm;
```

# Módulo 5 – WHILE (9 /9)

Por fim, vamos verificar como está o nosso código por completo. Você pode utilizar o Debug para ir observando todas essas variáveis e seus comportamentos durante a o loop de repetição.

# Módulo 5 – FOR ( 1/7)

A situação problema que vamos iniciar hoje é como percorrer um array e retornar a soma a cada elemento o seu índice.
Primeiro vamos aplicar com a estrutura de repetição **While** que aprendemos anteriormente.
Declare uma variável que armazenará o nosso contador de iteração e dentro da estrutura While, iremos declarar a condição de que o bloco de instrução deve ser executada.

Todo array possui uma propriedade que nos retorna o valor de seu comprimento que é o **.length**, basta colocar **nomeArray.length** para você ter o acesso a essa informação.

```
aula5_9 > JS arrayOperation.js > ...
1   const regularArray = [2, 5,
2
3   let count = 0;
4   while (count < regularArray.length)
```

```
(property) Array<number>.length: number
Gets or sets the length of the array. This is a number one higher than the highest index in the array.
```

E dentro do nosso bloco de instrução, iremos executar a **reatribuição** do valor **regularArray[count]**, ou seja, iteração por iteração (índice em índice), somando o valor do elemento com o valor do índice.
E para continuarmos a estrutura de repetição incrementamos mais uma iteração na nossa **variável count**, até a condição do **While** não ser mais contemplada.

```
1   const regularArray = [2, 5, 2, 7, 1, 3, 4, 7];
2
3   let count = 0;
4   while (count < regularArray.length){
5       regularArray[count] += count;
6       count++;
7   }
```

# Módulo 5 – FOR ( 2/7)

aula5_9 > JS arrayOperation.js > ...

```javascript
const regularArray = [2, 5, 2, 7, 1, 3, 4, 7];
//                    [2, 6, 4 , 10, 5, 8, 10, 14]
let count = 0;
while (count < regularArray.length) {
  regularArray[count] += count;
  count++;
}

console.log('O array atual é: ', regularArray);
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL SQL CONSOLE Filter (e.g. te

```
C:\Program Files\nodejs\node.exe .\aula5_9\arrayOperation.js
> O array atual é:  (8) [2, 6, 4, 10, 5, 8, 10, 14]
```

E para verificarmos o nosso programa, utilizaremos o **console.log()**, porém **não utilizaremos a string template**, porque ele converte o conteúdo que está dentro do **${}** e ele printa a string resultante. Na forma que iremos executar, estamos dizendo ao console.log para printar primeiro o texto e depois print o conteúdo desse array, assim preservamos **as estruturas** que são **mais complexas** como um array ou objeto.

Executamos o programa e vemos o retorno do nosso programa.

# Módulo 5 – FOR ( 3/7)

Nesse momento iremos aprender uma nova 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

}
```

# Módulo 5 – FOR ( 4/7)

Agora ao invés de utilizar o While, vamos alterar nosso código para que ele seja executado por um **FOR**.

```
1  const regularArray = [2, 5, 2, 7, 1, 3, 4, 7];
2
3  // let count = 0;
4  for (let count = 0; count < regularArray.length; count++) {
5    regularArray[count] += count;
6    //   count++;
7  }
8
9  console.log('O array atual é: ', regularArray);
```

Repare que trocamos a palavra While por FOR.

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.

É possível declarar a  variável controladora fora da **estrutura For** e depois atribuir outro valor dentro de suas declarações, mas geralmente já declaramos e definimos na própria estrutura, apenas em casos que você precise desse valor em outro lugar é recomendado declarar fora.

```
1  const regularArray = [2, 5, 2, 7, 1, 3, 4, 7];
2
3  let count = 254;
4  for (count = 0; count < regularArray.length; count++) {
5    regularArray[count] += count;
6    //   count++;
7  }
8
9  console.log('O array atual é: ', regularArray);
```

# Módulo 5 – FOR ( 5/7)

O **FOR** é muito parecido com o While. Caso você não queria declarar a primeira e a última declaração que o **FOR** pode utilizar, ou seja, o ponto de início da iteração e a incrementação. Você pode deixar as declarações inicial e final vazia, mas ainda precisa colocar o ponto e vírgula ( ; ) para sinalizar que as declarações estão vazias.

```
1  const regularArray = [2, 5, 2, 7, 1, 3, 4, 7];
2
3  let count = 0;
4  for (; count < regularArray.length; ) {
5    regularArray[count] += count;
6    count++;
7  }
8
9  console.log('O array atual é: ', regularArray);
```

Repare que se retirarmos os ponto e vírgulas ( ; ) o **For** fica parecido com a condicional do **While.**

Agora vamos **refatorar o nosso código do Fibonacci** e vamos utilizar a estrutura FOR para isso.

```
let currentIteration = 1;
```

Vamos atribuir o valor 1 á variável **currentIteration** e que depois armazenará o último elemento para o qual eu tenha a resposta.

Com a nova interpretação dessa variável, a nossa condição também mudará, pois queremos que quando a estrutura de repetição chegar a um determinado número, ele não realize mais uma iteração e sim que pare o bloco.

```
let currentIteration = 1;
while (currentIteration < termNumber) {
```

```
// if (termNumber === 1) {
//   return 1;
// }
```

Assim essa estrutura condicional não será mais útil, já que estamos inicializando com o nosso termo 1.

# Módulo 5 – FOR ( 7/7)

```
5     if (termNumber === 0) {
6       return 0;
7     }
8     // if (termNumber === 1) {
9     //   return 1;
10    // }
11    for (
12      let currentIteration = 1;
13      currentIteration < termNumber;
14      currentIteration++
15    ) {
16      let auxVar = lastTerm + secondToLast;
17      secondToLast = lastTerm;
18      lastTerm = auxVar;
19    }
20    return lastTerm;
21  }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL SQL CONSOLE

```
C:\Program Files\nodejs\node.exe .\aula5_9\fibonacciFor.js
O valor do termo número 6 da sequencia de fibonacci é 8
```

E para finalizar nosso código iremos transformar o While, com a estrutura do For declarando suas três informações inicial, condição e atualização.

Pronto! **A sequência de Fibonacci dessa vez utilizando a estrutura de repetição For.**

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 sai mesma. Uma função, por exemplo, pode solicitar outra invocação dela mesma para resolver um problema.

```
function calcularTemperaturaDaRuaHoje(dataDeHoje){

    ...

    calcularTemperaturaDaRuaHoje(dataDeOntem);
}
```

Observando através da Recursividade podemos entender o problema Fibonacci da seguinte forma:

Se um termo da sequência é definido a partir da soma dos termos anteriores, podemos dizer que ***" Fibo(numero) = Fibo(numero – 1) + Fibo (numero – 2)"***

0,1, **1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584**

A recursividade não é uma estrutura de repetição, mas sim um conceito, uma técnica que podemos utilizar para facilitar alguns processos.

Indo para o código de acordo com a nossa lógica, precisamos declarar os dois primeiros termos da sequência e criar uma condicional para os seus retornos. Dessa forma, a nossa função já tem a capacidade de retornar os termos 0 e 1.

```
if (termNumber === 0) {
  return 0;
}
if (termNumber === 1) {
  return 1;
}
```

Agora que nosso programa já tem o retornos dos primeiros dois termos, aplicaremos a técnica da Recursividade, então o nosso código terá essa estrutura.

```
1   function calculateFibonacciTerm(termNumber) {
2     if (termNumber === 0) {
3       return 0;
4     }
5     if (termNumber === 1) {
6       return 1;
7     }
8
9     return calculateFibonacciTerm(termNumber - 1) + calculateFibonacciTerm(termNumber -2);
10  }
11
12  console.log(
13    `O valor do termo número 6 da sequencia de fibonacci é ${calculateFibonacciTerm(
14      6
15    )}`
16  );
```

Caso não seja o termo 0 ou 1, ele irá executar duas chamadas dele menos, realizando a substituição da variável pelo termo que estamos buscando.

Agora que nosso programa já tem o retornos dos primeiros dois termos, aplicaremos a técnica da Recursividade, então o nosso código terá essa estrutura.

```
1   function calculateFibonacciTerm(termNumber) {
2     if (termNumber === 0) {
3       return 0;
4     }
5     if (termNumber === 1) {
6       return 1;
7     }
8
9     return calculateFibonacciTerm(termNumber - 1) + calculateFibonacciTerm(termNumber -2);
10  }
11
12  console.log(
13    `O valor do termo número 6 da sequencia de fibonacci é ${calculateFibonacciTerm(
14      6
15    )}`
16  );
```

Caso não seja o termo 0 ou 1, ele irá executar duas chamadas dele menos, realizando a substituição da variável pelo termo que estamos buscando.

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 possuí 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.

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.

# Módulo 5 – O problema da recusirvidade (2 / 2)

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.

**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.

# Módulo 5 – Enumerando informações em Objetos (2 / 3)

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

```javascript
const car = {
  brand: 'Toyota',
  model: 'Camry',
  year: 2023,
  color: 'Silver',
  fuelType: 'Gasoline',
  engineCapacity: '2.5L',
  transmission: 'Automatic',
  power: '203 horsepower',
  seatingCapacity: 5,
  price: '$25,000',
};

for (info in car) {
  console.log(info);
}
```

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.

# Módulo 5 – Enumerando informações em Objetos (3 / 3)

JS iteratingOnObject.js > ...

```javascript
const car = {
  brand: 'Toyota',
  model: 'Camry',
  year: 2023,
  color: 'Silver',
  fuelType: 'Gasoline',
  engineCapacity: '2.5L',
  transmission: 'Automatic',
  power: '203 horsepower',
  seatingCapacity: 5,
  price: '$25,000',
};

for (info in car) {
  console.log(`${info}: ${car[info]}`);
}
```

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.

# Módulo 5 – Enumerando informações em Arrays (1 / 3)

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.

index.js > 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.

# Módulo 5 – Enumerando informações em Arrays (3 / 3)

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.

# Módulo 5 – Valores Truthy e Falsy (1 / 2)

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".

```
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
```

# Módulo 5 – Valores Truthy e Falsy (2 / 2)

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 6

# DESENVOLVIMENTO WEB

DESENVOLVIMENTO WEB

DESENVOLVIMENTO WEB

# Módulo 6 – Como a comunicação da internet funciona (1/5)

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.

# Módulo 6 – Como a comunicação da internet funciona ( 4 / 5)

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 computadores 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 sejas 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 <img> 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 <br> é utilizada para quebrar uma linha de texto, sendo mais comum o uso da tag fechada <br />, 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.

## 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 <img>, links <a>, listas <ul> ou <ol>, 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.

O Vs Code possui uma ferramenta muito interessante para acompanharmos 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.

# Módulo 6 – O HTML ( 8 / 9)

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 > ...

```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>Document</title>
  </head>
  <body>
    <h1>Título da minha primeira página!</h1>
  </body>
</html>
```

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 navegarem 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**

- **<img>:** A tag <img> 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 <img> não possui uma tag de fechamento (</img>) 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 <img> 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 <img>. 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 <img> 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

```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>Document</title>
  </head>
  <body>
    <h2>Título da minha primeira página!</h2>
    <img src="./images/arara.jpg" alt="Arara sobrevoando uma floresta." />
    <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>
```

**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!

## 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.

```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>Document</title>
    <style>
      p {
        color: white;
        background-color: #3282f6;
      }
    </style>
  </head>
  <body>
    <h2>Título da minha primeira página!</h2>
    <img src="./images/arara.jpg" alt="Arara sobrevoando uma floresta." />
    <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;.

# Módulo 6 – Propriedades em CSS ( 3 / 3)

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

# Módulo 6 – Problemas dessa primeira abordagem(1/3)

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

```
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>
    <img src="./images/arara.jpg" alt="Arara sobrevoando u
    <p>
      Lorem ipsum dolor, sit amet consectetur adipisicing
      exercitationem asperiores possimus repudiandae volup
      assumenda amet. Officia aperiam commodi ducimus digr
      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.

# Módulo 6 – Problemas dessa primeira abordagem( 2 / 3)

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.

# Módulo 6 – Problemas dessa primeira abordagem( 3 / 3)

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.

Temos dois problemas acontecendo, o primeiro é que a tag style pode aumentar exponecialmente , 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 paras 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, ini
7      <title>Document</title>
8      <link rel="stylesheet" href="./style.css">
9    </head>
10   <body>
11     <h2>Título da minha primeira página!</h2>
12     <img src="./images/arara.jpg" alt="Arara sobrevoando u
13     <p>
14       Lorem ipsum dolor, sit amet consectetur adipisicing
15       exercitationem asperiores possimus repudiandae volup
16       assumenda amet. Officia aperiam commodi ducimus digr
17       quisquam ea, natus ipsum saepe pariatur!
18     </p>
19     <section>
20       <h1>Sobre JavaScript</h1>
21       <img src="./images/JS.png" alt="Logo do JavaScript."
22       <p>
23         JavaScript é a linguagem de programação mais impor
24         aprender a programar sistemas web
25       </p>
26     </section>
27   </body>
28 </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>
```

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

identifier expected css(css-identifierexpected)
<element class>
Selector Specificity: (0, 1, 0)
View Problem (Alt+F8)    No quick fixes available
.
```

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

**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;
}
```

## 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 <li> que são descendentes diretos ou indiretos de um elemento <ul>.

A regra **#div-conteudo-4-7 div {}** em CSS é um seletor que seleciona todos os elementos  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  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  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 <li> que são filhos diretos de um elemento <ul>.

A regra **#div-conteudo-4-7 > div {}** em CSS é um seletor que seleciona todos os elementos  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  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  é um descendente indireto, pois está dentro de outro elemento . Portanto, a regra não se aplica a ele.

Qualquer estilo definido dentro dessa regra será aplicado apenas aos elementos  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.

# Módulo 6 – O "BOX MODEL" (1 / 6)

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.

# Módulo 6 – O "BOX MODEL" (3 / 6)

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.

# Módulo 6 – O "BOX MODEL" (5 / 6)

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;
}
```

# Módulo 6 – O "BOX MODEL" (6 / 6)

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.

# Módulo 6 – Principais parâmetros estilizáveis (1 / 6)

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.

# Módulo 6 – Principais parâmetros estilizáveis (2 / 6)

Vamos iniciar com a estrutura do nosso arquivo index.html:

index.html > html > head > link

```html
<!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>
  </body>
</html>
```

E adicionaremos algumas estilizações na nossa página, para verificarmos o conceito do box model e de como ele age de tags pai/filho:

style.css — principais-propriedades

Welcome | index.html | style.css

style.css > ...

```css
body {
  padding: 30px;
  border: 3px solid red;
}
body > div {
  border: 3px solid blue;
}
```

Document

127.0.0.1:5500/index.html

Conteúdo da DIV

# Módulo 6 – Principais parâmetros estilizáveis (3 / 6)

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:

# Módulo 6 – Principais parâmetros estilizáveis (4 / 6)

Agora vamos adicionar mais um elemento em nossa estrutura HTML e observar como está sua estilização.

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.

# Módulo 6 – Principais parâmetros estilizáveis (5 / 6)

- **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.

# Módulo 6 – Principais parâmetros estilizáveis (6 / 6)

Exemplos da utilização da propriedade border-radius:

Módulo 7

# PRIMEIRO PROJETO

PRIMEIRO PROJETO

PRIMEIRO PROJETO

# Módulo 7 – Apresentando Projeto (1 / 3)

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.

# Módulo 7 – Apresentando Projeto (2 / 3)

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.

# Módulo 7 – Apresentando Projeto (3 / 3)

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:

# Módulo 7 – Recursos externos utilizados (2 / 8)

O site oficial da API mostra como que podemos utilizar ela:

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.

# Módulo 7 – Recursos externos utilizados (3 / 8)

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]&

# Módulo 7 – Recursos externos utilizados (4 / 8)

Acesse a aba API KEY

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 me 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:

**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:

# Módulo 7 – Recursos externos utilizados (8 / 8)

Clique em Code e faça o download do arquivo ZIP:

Depois iremos adicionar a pasta ao nosso projeto.

# Módulo 7 – 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.

# Módulo 7 – Começando a estrutura do projeto (3 / 7)

Iniciaremos a estrutura do 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>
```

# Módulo 7 – Começando a estrutura do projeto (6 / 7)

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.jsd
```

```
@import url("https://cdn.jsdelivr.net/npm/bo
```

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">
9    </head>
10   <body>
11     <header>MyMovies</header>
12     <main></main>
13   </body>
14 </html>
```

# Módulo 7 – Começando a estrutura do projeto (7 / 7)

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.

```
14    <header><i class="bi bi-film"></i>MyMovies</header>
```

# Módulo 7 – Criando o nosso Header (1 / 7)

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">
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>
```

**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.

```
<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.

```html
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>
```

# Módulo 7 – Criando o nosso Header (5 / 7)

Vamos agrupar os elementos do tipo label e input dentro de uma div com a classe="input-wrapper".

```html
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>
```

# Módulo 7 – Criando o nosso Header (6 / 7)

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    --ligthest-color: #e9ebd5;
5    --secondary-color: #774936;
6    --light-secondary: #c08972;
7  }
8
```

# Módulo 7 – Estilizando o Header (1 / 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 nosso 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></butto
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.

```
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  }
```

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.

# Módulo 7 – Estilizando o Header (7 / 8)

Nesse momento iremos fazer mais alguns ajustes de estilização do nosso Header, como cor, espaçamentos, tamanho dos elementos:

```
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(--ligthest-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  }
```

# Módulo 7 – Estilizando o Header (8 / 8)

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.

```html
<body>
  <header>
    <h2><i class="bi bi-film"></i>MyMovies</h2>
    <section id="search-container">
      <div class="input-wrapper movie-name-wrapper">
        <label for="movie-name">Nome:</label>
        <input type="text" id="movie-name" />
      </div>
      <div class="input-wrapper movie-year-wrapper">
        <label for="movie-year">Ano:</label>
        <input type="text" id="movie-year" />
      </div>
      <button><i class="bi bi-search"></i></button>
    </section>
  </header>
  <main></main>
</body>
```

```css
header > h2 > i {
  margin-right: 0.5rem;
}

.movie-name-wrapper {
  width: 65%;
}

.movie-year-wrapper {
  width: 30%;
}

#search-button {
  width: 5%;
}

.movie-year-wrapper > input {
  max-width: 5rem;
}
```

**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 7 – 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"**: representa a espessura normal da fonte, equivalente ao valor 400.
- **"bold"**: deixa a fonte em negrito, equivalente ao valor 700.
- **"bolder"**: deixa a fonte ainda mais grossa do que o valor atual.
- **"lighter"**: deixa a fonte mais fina do que o valor atual.

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.

```
.movie-name-wrapper {
  width: 65%;
  display: flex;
  align-items: center;
}

.movie-year-wrapper {
  width: 30%;
  display: flex;
  align-items: center;
}

#search-button {
  width: 5%;
}

.movie-year-wrapper > input {
  max-width: 5rem;
}

.movie-name-wrapper > input {
  width: 100%;
}

.movie-name-wrapper > input,
.movie-year-wrapper > input {
  margin: 0 0.5rem;
  border-radius: 7px;
  padding: 0.3rem 0.2rem;
  font-weight: bold;
}
```

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"**: define todas as propriedades de um elemento para seus valores iniciais padrão.
- **"inherit"**: faz com que todas as propriedades de um elemento herdem os valores de seu elemento pai.
- **"unset"**: remove todas as propriedades definidas anteriormente, restaurando-as para seus valores padrão ou herdados.

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(--ligthest-color);
76    border-radius: 10px;
77  }
78
79  #search-button:focus,
80  #search-button:hover {
81    background-color: var(--light-primary);
82  }
83
```

# Módulo 7 – Ajustando o Header para displays maiores (1 / 1)

Nesse momento iremos realizar o último ajuste do nosso Header para adaptar para display maiores ou menores.

```
#search-container {
  display: flex;
  align-items: center;
  width: 50%;
  justify-content: flex-end;
}

header > h2 > i {
  margin-right: 0.5rem;
}

.movie-name-wrapper {
  width: 65%;
  display: flex;
  align-items: center;
}

.movie-year-wrapper {
  /* width: 30%; */
  display: flex;
  align-items: center;
}
```

# Módulo 7 – Seção principal da página (1 / 5)

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 sobrescritos.

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).

# Módulo 7 – Seção principal da página (2 / 5)

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.

```
16  body {
17    display: flex;
18    flex-direction: column;
19    min-height: 100dvh;
20  }
```

# Módulo 7 – Seção principal da página (5 / 5)

A estilização do nosso main terá esse formato:

```
91  main {
92    text-align: center;
93    padding: 2rem 0;
94    background-color: var(--ligthest-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.

# Módulo 7 – Criando a estrutura do Modal (2 / 2)

A estrutura HMTL do nosso modal será criada com os elementos do código abaixo:

```html
<main>
  <h2>Comece agora a sua lista de filmes!</h2>
  <section id="movie-list"></section>
  <article id="modal-overlay">
    <section id="modal-background"></section>
    <section id="modal-container">
      <h2>Título do Filme</h2>
    </section>
  </article>
</main>
```

# Módulo 7 – Estilizando o Modal (1 / 7)

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.

```html
<head>
  <meta charset="UTF-8" />
  <meta http-equiv="X-UA-Compatible" content="IE=edge" /
  <meta name="viewport" content="width=device-width, ini
  <title>Lista de filmes para assistir</title>
  <link rel="stylesheet" href="./style.css" />
  <link
    rel="stylesheet"
    href="https://cdn.jsdelivr.net/npm/bootstrap-icons@1
  />
  <link rel="stylesheet" href="./modal.css" />
</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.

# Módulo 7 – Estilizando o Modal (4 / 7)

Aplicaremos agora no nosso modal as propriedades do CSS que acabamos de conhecer e algumas que já estamos familiarizados.

modal.css > ...

```
#modal-overlay {
    display: flex;
    position: fixed;
    top: 0;
    left: 0;
    width: 100vw;
    height: 100dvh;
    justify-content: center;
    align-items: center;
}
```

# Módulo 7 – Estilizando o Modal (5 / 7)

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 7 – Estilizando o Modal (7 / 7)

Você construirá a estrutura do nosso arquivo modal.css como o código abaixo.

```
# modal.css > #modal-container
:root {
  --primary-color: #29335c;
  --light-primary: #3f4f8d;
  --ligthest-color: #e9ebd5;
  --secondary-color: #774936;
  --light-secondary: #c08972;
}

#modal-overlay {
  display: flex;
  position: fixed;
  top: 0;
  left: 0;
  width: 100vw;
  height: 100dvh;
  justify-content: center;
  align-items: center;
}

#modal-background {
  background-color: var(--light-primary);
  width: 100%;
  height: 100%;
  position: absolute;
  z-index: 0;
  opacity: 0.4;
}

#modal-container {
  z-index: 1;
  background-color: var(--ligthest-color);
  padding: 1rem 3rem;
  border-radius: 10px;
}
```

# Módulo 7 – Mock do primeiro Modal (1 / 6)

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 7 – 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.

# Módulo 7 – Mock do primeiro Modal (3 / 6)


```
{
  "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 dynas
  "Language": "English, Italian, Latin",
  "Country": "United States",
  "Awards": "Won 3 Oscars. 32 wins & 31 nominations total"
  "Poster": "https://m.media-amazon.com/images/M/MV5BM2MyM
  "Ratings": [
    { "Source": "Internet Movie Database", "Value": "9.2/1
    { "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",
```


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.

# Módulo 7 – Mock do primeiro Modal (5 / 6)

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:

```html
<main>
  <h2>Comece agora a sua lista de filmes!</h2>
  <section id="movie-list"></section>
  <article id="modal-overlay">
    <section id="modal-background"></section>
    <section id="modal-container">
      <h2>Título do Filme</h2>
      <section id="modal-body"></section>
      <section id="modal-footer"></section>
    </section>
  </article>
</main>
```

# Módulo 7 – Mock do primeiro Modal (6 / 6)

```html
<main>
  <h2>Comece agora a sua lista de filmes!</h2>
  <section id="movie-list"></section>
  <article id="modal-overlay">
    <section id="modal-background"></section>
    <section id="modal-container">
      <h2>The Godfather - 1972</h2>
      <section id="modal-body">
        <img
          src="https://m.media-amazon.com/images/M/MV5
          alt="Poster do Filme."
        />
        <div id="movie-info">
          <p id="movie-plot">
            The aging patriarch of an organized crime
            York City transfers control of his clandes
            reluctant youngest son.
          </p>
          <p id="movie-cast">Marlon Brando, Al Pacino,
          <p id="movie-genre">Crime, Drama</p>
        </div>
      </section>
      <section id="modal-footer">
        <button id="add-to-list">Adicionar à Lista</bu
      </section>
    </section>
  </article>
</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.

# Módulo 7 – CSS do Modal projetado (3 / 4)

No arquivo style.css iremos aplicar algumas modificações que serão aplicadas a todos os botões do nosso projeto:

```
22  button {
23      all: unset;
24      cursor: pointer;
25  }
```

```
80  #search-button {
81      width: 5%;
82      padding: 0.3rem;
83      border: 2px solid var(--ligthest-color);
84      border-radius: 10px;
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 7 – 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:

```
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 #000;
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
```

# Módulo 7 – Adicionando inteligência ao Modal (1 / 6)

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');
```

# Módulo 7 – Adicionando inteligência ao Modal (3 / 6)

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**

```html
33      <article id="modal-overlay" class="overlay">
```

```html
64      </main>
65      <script src="./modal.js"></script>
66    </body>
```

**Arquivo CSS – modal.css**

```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**

```
.overlay {
  visibility: hidden;
  transition: all 0.3s ease;
  opacity: 0;
}

.open {
  visibility: visible;
  opacity: 1;
}
```

**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);
```

# Módulo 7 – Começando a inteligência da página principal (1 / 2)

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
```

# Módulo 7 – Começando a inteligência da página principal (2 / 2)

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 > ...
const searchButton = document.getElementById('search-button');
const overlay = document.getElementById('modal-overlay');

function searchButtonClickHandler() {
  overlay.classList.add('open');
}

searchButton.addEventListener('click', searchButtonClickHandler);
```

# Módulo 7 – Capturando as entradas do usuário (1 / 4)

Neste momento, vamos aprender como obter o retorno da nossa API. Para isso, vamos usar a URL 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  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.

# Módulo 7 – Capturando as entradas do usuário (3 / 4)

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.

# Módulo 7 – Capturando as entradas do usuário (4 / 4)



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.

# Módulo 7 – Consumindo a API do OMDB (2 / 5)

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 7 – 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);
```

# Módulo 7 – Consumindo a API do OMDB (4 / 5)

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.).

# Módulo 7 – Consumindo a API do OMDB (5 / 5)

## 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:

```
script.js
quero_assistir > 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.valuesplit(' ').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  }
```

# Módulo 7 – Lidando com erros (2 / 3)

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.

# Módulo 7 – Lidando com erros (3 / 3)

Fazendo os tratamentos de erros com os elementos de busca de Título do Filme e Ano, nosso código terá essa estrutura:

```javascript
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  }
```

```javascript
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 7 – 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:

```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>
    <link rel="stylesheet" href="./style.css" />
    <link
      rel="stylesheet"
      href="https://cdn.jsdelivr.net/npm/bootstrap-icons@1.10.5/font/bootstrap-icons.css"
    />
    <link rel="stylesheet" href="./modal.css" />
    <link rel="stylesheet" href="./src/notie-master/dist/notie.min.css" />
  </head>
```

```html
      </article>
    </main>
    <script src="./script.js"></script>
    <script src="./modal.js"></script>
    <script src="./key.js"></script>
    <script src="./src/notie-master/dist/notie.js"></script>
  </body>
</html>
```

## 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']
})
```

# Módulo 7 – Notificando o usuário de erros (4 / 4)

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');      You, 8 seconds ago • Uncommitted changes
  }
  overlay.classList.add('open');
} catch (error) {
  notie.alert({ type: 'error', text: error.message });
}
```

# Módulo 7 – Modal dinâmico (1 / 3)

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              <img
13                id="movie-poster"
14                src="https://m.media-amazon.com/images/M/MV5BM2MyNjYxNmUtYTAwNi00MTYxLWJmNWYtYzZl
15                alt="Poster do Filme."
16              />
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  }
```

# Módulo 7 – Modal dinâmico (2 / 3)

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
```

# Módulo 7 – Modal dinâmico (3 / 3)

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:

```
function createModal(data) {
  modalContainer.innerHTML = `
  <h2 id="movie-title">${data.Title} - ${data.Year}</h2>
          <section id="modal-body">
            <img
              id="movie-poster"
              src=${data.Poster}
              alt="Poster do Filme."
            />
            <div id="movie-info">
              <h3 id="movie-plot">
                ${data.Plot}
              </h3>
              <div id="movie-cast">
                <h4>Elenco:</h4>
                <h5>${data.Actors}</h5>
              </div>
              <div id="movie-genre">
                <h4>Gênero:</h4>
                <h5>${data.Genre}</h5>
              </div>
            </div>
          </section>
          <section id="modal-footer">
            <button id="add-to-list">Adicionar à Lista</button>
          </section>`;
}
```

```
JS script.js > searchButtonClickHandler
You, 1 second ago | 1 author (You)
const searchButton = document.getElementById('search-button');
const overlay = document.getElementById('modal-overlay');
const movieName = document.getElementById('movie-name');
const movieYear = document.getElementById('movie-year');

async function searchButtonClickHandler() {
  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');
    }
    createModal(data);
    overlay.classList.add('open');
  } catch (error) {      You, last week • initial commit …
    notie.alert({ type: 'error', text: error.message });
  }
}
```

# Módulo 7 – Preparando a lista de filmes (1 / 2)

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 deopis 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:

```
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
29      </section>
30    </header>
31    <main>
32      <h2>Comece agora a sua lista de filmes!</h2>
33      <section id="movie-list">
34        <article>
35          <img
36            src="https://m.media-amazon.com/images/M/MV5B
37            alt="Poster do filme."
38          />
39          <button>Remover</button>      You, 1 second ago
40        </article>
41      </section>
```

# Módulo 7 – 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.cs s:

```
#movie-list {
  display: flex;
  flex-wrap: wrap;
  justify-content: center;
  gap: 2rem;
  margin-top: 1rem;
}

#movie-list article {
  display: flex;
  flex-direction: column;
}

#movie-list article > img {
  border-radius: 10px;
  margin-bottom: 0.5rem;
}

.remove-button {
  border: 3px solid var(--secondary-color);
  padding: 0.3rem 0;
  width: 100%;
  border-radius: 10px;
  font-weight: bold;
  box-shadow: 1px 1px 10px #000;
  color: var(--secondary-color);
}

.remove-button:focus,
.remove-button:hover {
  background-color: var(--light-secondary);
  color: var(--ligthest-color);
}
```

# Módulo 7 – A inteligência do botão modal (1 / 5)

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  }
```

**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.

# Módulo 7 – A inteligência do botão modal (5 / 5)

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    }
```

# Módulo 7 – Finalizando a funcionalidade de adição (1 / 1)

Para finalizarmos a funcionalidade do botão de adicionar o filme, aplicaremos a inteligência para retornar os filmes de forma dinâmica.

```
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-${data.imdbID}'>
54    <img
55      src="${data.Poster}"
56      alt="Poster do ${data.Title}."
57    />
58    <button class="remove-button" onclick='{removeFilmFromList("${data.imdbID}")}'>
59      <i class="bi bi-trash"></i> Remover
60    </button>
61  </article>`;
62  }
63
```

# Módulo 7 – 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-${data.imdbID}'>
  <img
    src="${data.Poster}"
    alt="Poster do ${data.Title}."
  />
  <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.

# Módulo 7 – Identificando os filmes de forma única (2 / 2)

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.

```
function addToList(data) {
  if (isFilmAlreadyOnTheList(data.imdbID)) {
    notie.alert({ type: 'error', text: 'Filme já está na lista' });
    return;
  }
  movieList.push(data);
  updateUI(data);
  overlay.classList.remove('open');
}
```

# Módulo 7 – Remoção de um filme (1 / 3)

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();
}
```

# Módulo 7 – Remoção de um filme (2 / 3)

```
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.

# Módulo 7 – Remoção de um filme (3 / 3)



Finalizando nosso botão de remover vamos aplicar a inteligência ao elemento do botão do HTML.

```
function updateUI(data) {
  movieListContainer.innerHTML += `<article id='movie-card-${data.imdbID}'>
  <img
    src="${data.Poster}"
    alt="Poster do ${data.Title}."
  />
  <button class="remove-button" onclick='{removeFilmFromList("${data.imdbID}")}'>
    <i class="bi bi-trash"></i> Remover
  </button>
</article>`;
}
```

# Módulo 7 – Ajustes finais / armazenando a lista no navegador (1 / 8)

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:

```
#movie-list article {
  display: flex;
  flex-direction: column;
  justify-content: space-between;
}
```

Aplicar um tamanho fixo ao elemento de ícone da busca:

```
#search-button {
  width: 20px;
  padding: 0.3rem;
  border: 2px solid var(--ligthest-color);
  border-radius: 10px;
}
```

# Módulo 7 – 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;
67  }
```

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.

**Operado 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')) ?? [];
```

# Módulo 7 – Ajustes finais / armazenando a lista no navegador (6 / 8)

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 change
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  }
```

# Módulo 7 – Ajustes finais / armazenando a lista no navegador (7 / 8)

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:

```
82  movieList.forEach(updateUI);
```

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.

Módulo 8

# 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.

# Módulo 8 – Antecipando erros com condicionais (1 / 5)

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).

index.html > ...

```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, ini
    <title>Pythagoras' Web Page</title>
    <link rel="stylesheet" href="style.css" />
  </head>
  <body>
    <h2>Calcule a Hipotenusa de qualquer Triângulo Retângu
    <div class="answer-container">
      <img
        src="assets/pythagoras.jpeg"
        alt="Busto de Pitágoras."
        id="pythagoras"
      />
      <div id="answer" class="hidden-element"></div>
    </div>
    <div class="inputs-container">
      <label for="side-1">Primeiro Cateto</label>
      <input id="side-1" type="text" />
      <label for="side-2">Segundo Cateto</label>
      <input id="side-2" type="text" />
      <button id="calculate-hypotenuse">Calcular Hipotenus
    </div>
    <script src="/calculatePythagoras.js"></script>
```

calculatePythagoras.js > onCalculateButtonClick

```js
function calculateSquareRoot(radicand) {
  return radicand ** (1 / 2);
}

function calculatePythagoras(side1, side2) {
  const sum = side1 ** 2 + side2 ** 2;
  return calculateSquareRoot(sum);
}

const firstSideRef = document.getElementById('side-1');
const secondSideRef = document.getElementById('side-2');
const calculateButton = document.getElementById('calculate
const answerDiv = document.getElementById('answer');

function onCalculateButtonClick() any
  const firstSide = firstSideRef.value;
  const secondSide = secondSideRef.value;
  // console.log('function executed');
  answerDiv.classList.remove('hidden-element');
  answerDiv.innerHTML = `A hipotenusa de um triangulo de l
    firstSide,
    secondSide
  )
    .toFixed(3)
    .replace(/[.,]000$/, '')}
  <div class="left-point"></div>`;
}
```

# Módulo 8 – Antecipando erros com condicionais (2 / 5)



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
}
```

# Módulo 8 – Try, Catch e Throw (2 / 5)

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.

# Módulo 8 – Try, Catch e Throw (4 / 5)

```
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);
  }
}
```

# Módulo 8 – Try, Catch e Throw (5 / 5)

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(radicand) {
  try {
    if (radicand < 0) {
      throw 'O radicando deve ser real e positivo';
    }
    return radicand ** (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);
  }
}
```

# Módulo 8 – Lançar erro completo ou só a mensagem ? (1 / 2)

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.

# Módulo 8 – Lançar erro completo ou só a mensagem ? (2 / 2)

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                                   calculatePythagoras.js:13
    at calculateSquareRoot (calculatePythagoras.js:3:11)
    at calculatePythagoras (calculatePythagoras.js:11:12)
    at HTMLButtonElement.onCalculateButtonClick (calculatePythagoras.js:37:100)
TypeError: Cannot read properties of undefined (reading 'toFixed')            calculatePythagoras.js:45
    at HTMLButtonElement.onCalculateButtonClick (calculatePythagoras.js:41:7)
```

## 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.

# Módulo 8 – Finally (2 / 2)

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   calculatePythagoras.js:45
  números positivos exclusivamente!
      at HTMLButtonElement.onCalculateButtonClick (calculatePythagoras.js:32:13)
  bloco finally executado!                                                    calculatePythagoras.js:47
>
```

Módulo 9

# JAVASCRIPT ASSÍNCRONO

JAVASCRIPT ASSÍNCRONO

JAVASCRIPT ASSÍNCRONO

**Sincronia**

A sincronia em programação JavaScript refere-se à forma como as operações são executadas em sequência, uma após a outra. Quando uma operação é executada de forma síncrona, o programa aguarda a conclusão dessa operação antes de prosseguir para a próxima.

Em JavaScript, por padrão, as operações são executadas de forma síncrona. Isso significa que o código é executado linha por linha, e cada linha de código é executada completamente antes de passar para a próxima linha. Por exemplo, se houver uma função sendo chamada, o programa aguardará a conclusão dessa função antes de continuar.
A sincronia é útil em muitos casos, pois garante que as operações sejam executadas na ordem correta e que o resultado de uma operação esteja disponível antes de ser usado em outra operação.

```
JS console_test_code.js ×
C: > Users > danie > JS console_test_code.js
1   console.log(1);
2   console.log(2);
3   console.log(3);
4
```

```
C:\Program Files\nodejs\node.exe .\console_test_code.js
1
2
3
```

# Módulo 9 – Introdução / programação síncrona (2 / 4)

No entanto, a sincronia também pode ter algumas desvantagens. Se uma operação levar muito tempo para ser concluída, ela pode bloquear a execução de outras operações, tornando o programa lento e não responsivo. Isso é especialmente problemático em aplicações web, onde a interação do usuário pode ser afetada.

Um exemplo disso é um código síncrono de longa duração encontrado no site da Mozilla (MDN). Esse código usa um algoritmo ineficiente para gerar vários números primos grandes quando o usuário clica no botão "Gerar primos". Quanto maior for o número de primos especificado pelo usuário, mais tempo a operação levará.

script.js > ...

```js
const MAX_PRIME = 1000000;

function isPrime(n) {
  for (let i = 2; i <= Math.sqrt(n); i++) {
    if (n % i === 0) {
      return false;
    }
  }
  return n > 1;
}

const random = (max) => Math.floor(Math.random() * max);

function generatePrimes(quota) {
  const primes = [];
  while (primes.length < quota) {
    const candidate = random(MAX_PRIME);
    if (isPrime(candidate)) {
      primes.push(candidate);
    }
  }
  return primes;
}

const quota = document.querySelector('#quota');
const output = document.querySelector('#output');

document.querySelector('#generate').addEventListener('click', () => {
  const primes = generatePrimes(quota.value);
  output.textContent = `Finished generating ${quota.value} primes!`;
});
```

index.html > html > head > link

```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>Document</title>
    <link rel="stylesheet" href="./style.css" />
  </head>
  <body>
    <label for="quota">Number of primes:</label>
    <input type="text" id="quota" name="quota" value="1000000" />

    <button id="generate">Generate primes</button>
    <button id="reload">Reload</button>

    <div id="output"></div>
    <script src="./script.js"></script>
  </body>
</html>
```

# Módulo 9 – Introdução / programação síncrona (3 / 4)

Temos um programa que gera números primos grandes quando o usuário clica em um botão chamado "Gerar primos". No entanto, o algoritmo usado para gerar esses números primos é muito ineficiente. Isso significa que, quanto maior for o número de primos que o usuário especificar, mais tempo o programa levará para gerá-los.

Vamos supor que o usuário queira gerar 100 números primos. O programa começará a gerar esses números, mas como o algoritmo é ineficiente, ele levará um tempo considerável para encontrar cada número primo. Durante esse tempo, o programa ficará "travado", sem responder a outras interações do usuário.

Isso acontece porque o programa está executando as operações de forma síncrona. Ou seja, ele aguarda a conclusão de cada operação antes de prosseguir para a próxima. Nesse caso, ele aguarda a geração de cada número primo antes de passar para o próximo.

**setTimeout()**

A função setTimeout() é uma função embutida em JavaScript que permite agendar a execução de uma função após um determinado período de tempo. Ela é comumente usada para criar atrasos ou agendar tarefas para serem executadas no futuro.

A sintaxe básica da função setTimeout() é a seguinte:

```
setTimeout(funcao, tempo);
```

- **funcao** é a função que será executada após o tempo especificado.
- **tempo** é o tempo em milissegundos que deve ser aguardado antes de executar a função.

```
JS consoleLogging.js > ...
1   console.log(1);
2   setTimeout(write2OnConsole, 2000);
3   console.log(3);
4
5   function write2OnConsole() {
6     console.log(2);
7   }
8
```

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_9\aula_1> node .\consoleLogging.js
1
3
```

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_9\aula_1> node .\consoleLogging.js
1
3
2
```

**Programação Assíncrona**

A programação assíncrona em JavaScript é uma abordagem que permite que o código execute tarefas de forma não sequencial, ou seja, sem precisar esperar a conclusão de uma tarefa antes de iniciar outra. Isso é especialmente útil quando lidamos com operações que podem levar algum tempo, como fazer requisições de rede, ler arquivos ou executar consultas em bancos de dados.

Em vez de bloquear a execução do código enquanto aguarda a conclusão de uma tarefa, a programação assíncrona permite que o programa continue a responder a outras interações do usuário ou executar outras tarefas enquanto aguarda a conclusão da tarefa assíncrona.

Em JavaScript, existem várias maneiras de lidar com a programação assíncrona. Alguns dos principais recursos e técnicas incluem:

- **Callbacks:** Os callbacks são funções que são passadas como argumentos para outras funções assíncronas. Quando a tarefa assíncrona é concluída, a função de callback é chamada para lidar com o resultado. Isso permite que o programa continue a executar outras tarefas enquanto aguarda a conclusão da tarefa assíncrona.

- **Promises:** As Promises são objetos que representam o resultado futuro de uma operação assíncrona. Elas permitem que você encadeie várias operações assíncronas de forma mais legível e organizada. As Promises têm métodos como then() e catch() para lidar com o resultado ou o erro da operação assíncrona.

- **Async/await:** O async/await é uma sintaxe mais moderna para lidar com a programação assíncrona em JavaScript. Ele permite que você escreva código assíncrono de forma mais síncrona, tornando-o mais fácil de ler e entender. A palavra-chave async é usada para definir uma função assíncrona, e a palavra-chave await é usada para esperar a conclusão de uma operação assíncrona dentro dessa função.

# Módulo 9 – Programação assíncrona (2 / 4)

A programação assíncrona é especialmente útil em aplicações web, onde a interação do usuário é importante. Ela permite que o programa continue a responder a outras interações do usuário enquanto aguarda a conclusão de operações demoradas, tornando a aplicação mais responsiva e melhorando a experiência do usuário.

Para lidar com o problema visto na aula passada, o JavaScript então oferece a possibilidade de executar operações de forma assíncrona. Isso significa que o programa pode iniciar uma operação e continuar a execução sem esperar pela conclusão dessa operação. Quando a operação assíncrona é concluída, uma função de retorno (callback) é chamada para lidar com o resultado.

Resumindo, a programação assíncrona é  útil quando se lida com operações que podem levar algum tempo, como fazer requisições de rede, ler arquivos ou executar consultas em bancos de dados. Ela permite que o programa continue a responder a outras interações do usuário enquanto aguarda a conclusão dessas operações assíncronas.

# Módulo 9 – Programação assíncrona (3 / 4)

Vou dar alguns exemplos simples de situações em que a programação assíncrona é utilizada na computação, como o processo de impressão e a comunicação pela internet.

- **Processo de impressão:** Imagine que você envia um documento para ser impresso em uma impressora. O processo de impressão pode levar algum tempo, especialmente se o documento for grande. Em vez de bloquear o seu computador e esperar a impressão ser concluída antes de poder fazer outras tarefas, a programação assíncrona permite que você continue a usar o computador normalmente enquanto a impressão está em andamento. Assim, você pode abrir outros aplicativos, navegar na internet ou realizar outras tarefas enquanto a impressão é processada em segundo plano.

- **Comunicação pela internet:** Quando você faz uma requisição para acessar um site ou enviar uma mensagem pela internet, essa comunicação ocorre de forma assíncrona. Por exemplo, ao enviar um e-mail, você não precisa esperar que o e-mail seja enviado antes de poder fazer outras coisas no seu computador. A programação assíncrona permite que a comunicação com o servidor de e-mail ocorra em segundo plano, enquanto você continua a usar o seu computador normalmente. Isso também se aplica a outras formas de comunicação pela internet, como fazer uma requisição para obter dados de um servidor ou enviar uma mensagem em um aplicativo de bate-papo.

# Módulo 9 – Programação assíncrona (4 / 4)

Conseguimos verificar isso quando utilizamos o setTimeout e tentamos imprimir o seu retorno, por ser uma função assíncrona o nosso programa nos retornará a seguinte saída:

```
JS consoleLogging.js > write2OnConsoleAndReturnString
1   console.log(1);
2   const result = setTimeout(write2OnConsoleAndReturnString, 2000);
3   console.log(result);
4
5   function write2OnConsoleAndReturnString() {
6     console.log(2);
7     return 'Procedure Completed';
8   }
9
```

```
C:\Program Files\nodejs\node.exe .\consoleLogging.js
1
> Timeout {_idleTimeout: 2000, _idlePrev: TimersList, _idleNext: TimersList, _idleStart: 186, _onTimeout: ƒ, …}
2
```

# Módulo 9 – Funções anônimas (1 / 3)

Vamos abordar um conceito novo de função nesse momento, que será muito útil para progredirmos com o contexto de assincronicidade. Inicialmente temos uma código de uma função que tem como objetivo imprimir no terminal uma mensagem.

```
JS functions.js > ...
1  function writeThisMessageOnConsole(message) {
2    console.log(message);
3  }
4
5  writeThisMessageOnConsole('Escrevi a mensagem com sucesso!');
6
```

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_9\aula_3> node ./functions.js
Escrevi a mensagem com sucesso!
```

Até esse momento, escrevemos uma função que retorna a mensagem no terminal. Vamos explorar essa função com o setTimeout() que aprendemos anteriormente.

```
JS functions.js > ...
1  writeThisMessageOnConsole('Escrevi a mensagem com sucesso!');
2
3  setTimeout(writeThisMessageOnConsole, 2000);
4
5  function writeThisMessageOnConsole(message) {
6    console.log(message);
7  }
8
```

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_9\aula_3> node ./functions.js
Escrevi a mensagem com sucesso!
undefined
```

Repare que houve dois retornos no nosso terminal, a mensagem e o tipo undefined, mas o que houve?

# Módulo 9 – Funções anônimas (2 / 3)

Vamos modificar nosso código e verificar como estamos retornando a nossa função e o que a setTimeout() irá mostrar no terminal dessa vez.

```
functions.js > ...
setTimeout(writeThisMessageOnConsole('Escrevi a mensagem com sucesso!'), 2000);

function writeThisMessageOnConsole(message) {
  console.log(message);
}
```

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_9\aula_3> node ./functions.js
Escrevi a mensagem com sucesso!
node:internal/validators:421
    throw new ERR_INVALID_ARG_TYPE(name, 'Function', value);
    ^

TypeError [ERR_INVALID_ARG_TYPE]: The "callback" argument must be of type function. Received undefined
    at setTimeout (node:timers:141:3)
    at Object.<anonymous> (C:\Users\danie\JavaScript Impressionador - arquivos\modulo_9\aula_3\functions.
    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)
```

Nesse momento o nosso código retorna um erro.
O erro "invalid_arg_type - return undefined" em JavaScript ocorre quando uma função é chamada com um argumento inválido e não retorna um valor definido, ou seja, a nossa função não está retornando nenhum valor, e quando utilizamos o setTimeout estamos utilizando o retorno dessa função, que nesse momento é undefined.
Para observar esse comportamento vamos alterar o nosso código e ver o retorno que está ocorrendo.

```
functions.js > result
// setTimeout(writeThisMessageOnConsole('Escrevi a mensagem com sucesso!'), 2000);

function writeThisMessageOnConsole(message) {
  console.log(message);
  return;
}

const result = writeThisMessageOnConsole('parametro');
console.log(result);
```

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_9\aula_3> node ./functions.js
parametro
undefined
```

Conseguimos verificar que quando utilizamos ou não o return; sabemos que a nossa função não estará retornando nada, ou seja, automaticamente o seu tipo será undefined.

**Função anônima**

Uma função anônima em JavaScript é uma função que não possui um nome definido. Em vez disso, ela é atribuída a uma variável ou passada como argumento para outra função. As funções anônimas são úteis quando você precisa de uma função temporária ou quando deseja criar uma função de forma mais concisa. Elas são frequentemente usadas em callbacks, onde uma função é passada como argumento para outra função e é executada posteriormente.

```
JS functions.js > ...
1   setTimeout(function () {
2     console.log('Função que eu defini dentro do setTimeout');
3   }, 2000);
```

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_9\aula_3> node ./functions.js
Função que eu defini dentro do setTimeout
```

O que está acontecendo no código acima é que a setTimeout() agendará a execução da função anônima após 2 segundos e, quando executada, chamará a função writeThisMessageOnConsole passando a mensagem "Escreva essa minha mensagem aqui!" como argumento.

# Módulo 9 – Armazenando funções em variáveis (1 / 1)

Em JavaScript, você pode armazenar uma função anônima em uma variável atribuindo-a diretamente à variável.

```
function writeThisMessageOnConsole(message) {
  console.log(message);
}

const resultFunction = function () {
  writeThisMessageOnConsole('Escreva essa minha mensagem aqui!');
};
```

A função anônima é atribuída à variável resultFunction. A função não possui um nome, mas pode ser chamada posteriormente usando o nome da variável seguido pelos parênteses.

```
resultFunction();
```

```
PS C:\Users\danie\JavaScript Impressionador - arquivos\modulo_9\aula_4> node .\functions.js
Escreva essa minha mensagem aqui!
```

**Arrow Functions**

Uma arrow function, ou função de seta em português, é uma forma mais concisa de escrever funções em JavaScript. Ela foi introduzida na versão ES6 (ECMAScript 2015) da linguagem e oferece algumas vantagens em relação às funções tradicionais.

Uma das principais vantagens das arrow functions é a sintaxe mais curta e concisa. Se a função tiver apenas uma expressão de retorno, você pode omitir as chaves e a palavra-chave return. A sintaxe da arrow function é composta por uma seta ( ) que separa os parâmetros da função do corpo da função.

```
const writeUserMessageArrowVersion = (message) => {
  console.log(message);
};

writeUserMessageArrowVersion(
  'Escreva esse argumento no console - Versão Flecha ~ (arrow function)'
);
```

# Módulo 9 – Arrow Functions (2 / 2)

O código abaixo nos mostra um exemplo de arrow function anônima, e para conseguirmos executá-la precisamos atribuir a mesma á uma variável.

```
() => {
  console.log('essa é uma função!');
};
```

```
const variavelQueAbrigaArrowFunction = () => {
  console.log('essa é uma função!');
};

variavelQueAbrigaArrowFunction()
```

# Módulo 9 – As diferentes formas de uma Arrow Function (1 / 2)

No exemplo abaixo, a função myLoggingFunction é uma arrow function que recebe um parâmetro message e imprime esse valor no console usando console.log(). Ela pode ser usada para facilitar o registro de mensagens no console durante a execução do código.

```
JS different_arrow.js > ...
1  const myLoggingFunction = (message) => {
2    console.log(message);
3  };
4
5  myLoggingFunction('Teste da primeira arrow function do arquivo');
```

Perceba que atribuimos a variável a nossa função e depois executamos ela, agora Se você atribuir uma função JavaScript a uma variável e tentar chamá-la ou executá-la antes de atribuí-la, isso resultará em um erro. Isso ocorre porque, antes de atribuir uma função a uma variável, a variável não possui um valor definido e, portanto, não pode ser chamada como uma função.

Portanto, é importante garantir que as funções sejam atribuídas a variáveis antes de serem chamadas ou executadas para evitar erros de "is not a function".

```
C:\Program Files\nodejs\node.exe .\different_arrow.js
Process exited with code 1
Uncaught ReferenceError ReferenceError: Cannot access 'myNormalLoggingFunction' before initialization
    at <anonymous> (c:\Users\danie\JavaScript Impressionador - arquivos\modulo_9\aula_6\different_arrow.js:5:1)
    at Module._compile (internal/modules/cjs/loader:1218:14)
    at Module._extensions..js (internal/modules/cjs/loader:1272:10)
    at Module.load (internal/modules/cjs/loader:1081:32)
    at Module._load (internal/modules/cjs/loader:922:12)
    at executeUserEntryPoint (internal/modules/run_main:81:12)
    at <anonymous> (internal/main/run_main_module:23:47)
```

# Módulo 9 – As diferentes formas de uma Arrow Function (2 / 2)

Se você possui apenas um parâmetro em sua arrow function, não é necessário colocá-lo entre parênteses. Você pode escrever o parâmetro diretamente após a seta (=>) para tornar o código mais conciso. E se você tiver mais de um parâmetro ou nenhum parâmetro, precisará usar parênteses para separá-los.

```
const myLoggingFunctionWithOnlyOneParameter = message => {
  console.log(message);
};
```

Temo um outro cenário que ao escrever uma arrow function com apenas uma instrução, você pode optar por não usar as chaves {} para definir um bloco de instrução. No entanto, se você tiver mais de uma instrução, será necessário usar as chaves para definir um bloco de instrução.

```
const myLoggingFunctionWithOnlyOneInstruction = (message, message2) => console.log(message + message2);
```

# Módulo 9 – Callbacks (1 / 5)

Para usar valores de operações assíncronas, primeiro precisamos entender o conceito de programação assíncrona. Em alguns casos, é necessário permitir que certas funções ou partes do código sejam executadas de forma assíncrona, ou seja, sem seguir a sequência normal de execução.

Quando uma operação assíncrona é executada, ela não bloqueia a execução do restante do código. Em vez disso, ela é executada em segundo plano e, quando concluída, pode retornar um valor ou executar uma determinada ação.

JS consoleLogging.js > ...

```js
console.log(1);
const result = setTimeout(write2OnConsoleAndReturnString, 2000);
console.log(result);

function write2OnConsoleAndReturnString() {
  console.log(2);
  return 'Procedure Completed';
}

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL SQL CONSOLE Filt

```
C:\Program Files\nodejs\node.exe .\consoleLogging.js
1
> Timeout {_idleTimeout: 2000, _idlePrev: TimersList, _idleNext: TimersList, _idleStart: 196, _onTimeout: ƒ, …}
2
```

Para utilizar corretamente os valores obtidos de uma operação assíncrona, geralmente usamos callbacks, promessas ou async/await. Esses mecanismos nos permitem lidar com a assincronicidade de forma mais eficiente e organizada.

## CALLBACKS

Em JavaScript, um callback é uma função que é passada como argumento para outra função e é executada quando um determinado evento ocorre ou quando uma operação assíncrona é concluída. O objetivo do callback é permitir que o código seja executado de forma assíncrona, sem bloquear a execução do restante do programa.
Aqui está um exemplo simples de como usar callbacks em JavaScript:

```javascript
function fazerAlgo(callback) {
  // Simulando uma operação assíncrona
  setTimeout(function() {
    // Executando o callback após 2 segundos
    callback("Resultado da operação");
  }, 2000);
}

function callbackFuncao(resultado) {
  console.log("O resultado da operação é: " + resultado);
}

// Chamando a função fazerAlgo e passando o callback como argumento
fazerAlgo(callbackFuncao);
```

Neste exemplo, temos a função fazerAlgo que simula uma operação assíncrona usando setTimeout. Ela recebe um callback como argumento e, após 2 segundos, chama esse callback passando o resultado da operação.

A função callbackFuncao é o callback que passamos para fazerAlgo. Quando a operação assíncrona é concluída, o callback é executado e recebe o resultado como parâmetro. Neste caso, o callback simplesmente imprime o resultado no console.

Ao chamar fazerAlgo(callbackFuncao), estamos passando a função callbackFuncao como argumento para fazerAlgo. Quando a operação assíncrona é concluída, o callback é executado e o resultado é impresso no console.

# Módulo 9 – Callbacks (3 / 5)

Callbacks são amplamente utilizados em JavaScript para lidar com operações assíncronas, como requisições AJAX, leitura de arquivos, chamadas de API, entre outros. Eles permitem que o código seja executado de forma não sequencial, garantindo que as operações assíncronas sejam tratadas corretamente.

As funções de callback nos permitem garantir um comportamento sincronizado em operações assíncronas, já que essas funções só serão executadas quando o processo assíncrono for concluído.

É importante lembrar que callbacks podem ser aninhados, o que pode levar a um código complexo e difícil de ler. Para evitar esse problema, surgiram outras abordagens, como Promises e async/await, que fornecem uma sintaxe mais limpa e organizada para lidar com operações assíncronas em JavaScript.

Vamos para mais um exemplo, seguindo o código que estamos construindo nesse módulo:

```
const result = setTimeout(() => write2OnConsoleAndReturnString(console.log), 2000);

function write2OnConsoleAndReturnString(callbackFunction) {
  console.log(2);
  callbackFunction('Procedure Completed');
}
```

Neste código, temos uma função chamada write2OnConsoleAndReturnString que recebe um callback como parâmetro. Essa função imprime o número 2 no console e, em seguida, chama o callback passando a string 'Procedure Completed' como argumento.

Em seguida, temos a linha const result = setTimeout(() => write2OnConsoleAndReturnString(console.log), 2000);. Aqui, estamos usando a função setTimeout para simular uma operação assíncrona que será executada após 2 segundos.

Dentro de setTimeout, temos uma função de seta () => write2OnConsoleAndReturnString(console.log). Essa função de seta é o callback que será executado após o tempo especificado (2 segundos) pelo setTimeout.

# Módulo 9 – Callbacks (5 / 5)

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

```
const result = setTimeout(() => write2OnConsoleAndReturnString(console.log), 2000);

function write2OnConsoleAndReturnString(callbackFunction) {
  console.log(2);
  callbackFunction('Procedure Completed');
}
```

Dentro do callback, estamos chamando a função write2OnConsoleAndReturnString e passando console.log como argumento. Isso significa que, quando o callback for executado, a função write2OnConsoleAndReturnString será chamada e o console.log será usado como o callback interno dessa função.

Dentro de write2OnConsoleAndReturnString, primeiro imprimimos o número 2 no console usando console.log(2). Em seguida, chamamos o callback passado como parâmetro, que é console.log, e passamos a string 'Procedure Completed' como argumento.

```
C:\Program Files\nodejs\node.exe .\consoleLogging.js
1
2
Procedure Completed
```

# Módulo 9 – Passar função como argumento VS. Chamar função (1 / 4)

Nesse momento, vamos verificar como que um função é lida pelo Javascript. Podemos fazer a definição e atribuição de uma funcionalidade para uma função, como no exemplo abaixo:

```
> function procedimento1(){
      console.log('procedimento 1');
  }
<· undefined
> procedimento1()
  procedimento 1
<· undefined
> function procedimento1(){
      console.log('Essa funcao escreveu procedimento 1 no console com sucesso');
  }
<· undefined
> procedimento1()
  Essa funcao escreveu procedimento 1 no console com sucesso
<· undefined
>
```

Nesse exemplo, criamos um função chamada procedimento1() e atribuímos uma funcionalidade de imprimir no terminal através do console.log a mensagem 'procedimento 1'.
Chamamos a função e verificamos que a sua execução foi feita com sucesso.

Para observarmos novamente esse comportamento, redefinimos a mensagem que será impressa e chamamos novamente a função.

Até agora observamos o comportamento da função como estamos utilizando ao longo do curso.

# Módulo 9 – Passar função como argumento VS. Chamar função (2 / 4)

Mas como que o Javascript entende a função quando não chamamos para executar? Qual é o retorno da função sem ela ter sido executada?

Podemos observar ao lado, que ao chamar a função procedimento1() o Javascript retorna a sua execução, porém ao invocá-la apenas com o seu nome procedimento1 ele nos retorna a descrição de um objeto.

Porque em JavaScript, uma função é um tipo especial de objeto. Isso significa que, assim como outros objetos em JavaScript, uma função pode ter propriedades e métodos.

Essa natureza de objeto das funções em JavaScript permite que elas sejam flexíveis e poderosas, podendo ser tratadas como qualquer outro objeto e podendo ser passadas como argumentos, armazenadas em variáveis e ter propriedades e métodos definidos.

```
> procedimento1()
  Essa funcao escreveu procedimento 1 no console com sucesso
<· undefined
> procedimento1
<· ƒ procedimento1(){
      console.log('Essa funcao escreveu procedimento 1 no console com sucesso');
  }
```

# Módulo 9 – Passar função como argumento VS. Chamar função (3 / 4)

**Passar uma função como argumento:**

Em JavaScript, é possível passar uma função como argumento para outra função. Isso significa que você pode fornecer uma função como um valor para ser usado dentro de outra função. Essa técnica é conhecida como "callback" e é amplamente utilizada em JavaScript para lidar com operações assíncronas, eventos e outras situações em que é necessário executar um código em resposta a um evento específico.

functions_reinforcement.js > ...

```javascript
function procedimento1(outraFuncao) {
  //
  outraFuncao();
}

function procedimento2() {
  console.log('a função procedimento 2 foi executada com sucesso');
  return 'Banana';
}

//procedimento1('Banana') = procedimento1(procedimento2())

procedimento1(procedimento2);
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL SQL CONSOLE

```
C:\Program Files\nodejs\node.exe .\functions_reinforcement.js
a função procedimento 2 foi executada com sucesso
```

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

var resultado = somar(2, 3);
console.log(resultado); // 5
```

**Chamar uma função:**

Chamar uma função em JavaScript significa executar o código dentro da função. Isso é feito usando o nome da função seguido por parênteses (). Quando uma função é chamada, o código dentro dela é executado e qualquer valor de retorno é retornado.

Neste exemplo, a função somar é chamada com os argumentos 2 e 3. O código dentro da função é executado, realizando a soma dos dois números e retornando o resultado. Esse resultado é atribuído à variável resultado e, em seguida, é impresso no console.

Em resumo, passar uma função como argumento envolve fornecer uma função como valor para ser usada dentro de outra função, enquanto chamar uma função envolve executar o código dentro da função e obter um valor de retorno, se houver. Ambas as técnicas são úteis em diferentes situações e desempenham papéis importantes na programação em JavaScript.

# Módulo 9 – Callback Hell (1 / 6)

Imagine um cenário no qual um Usuário irá entrar em uma Rede Social, através de um aplicativo, e esse usuário irá pesquisar um amigo de sua lista de amigos, e ao clicar no amigo selecionado o aplicativo retorna as fotos desse amigo.

1. Vamos analisar esse cenário, estamos dizendo que iremos fazer uma requisição ao nosso Back End que irá retorna primeiro a resposta do nosso login, ou seja, ele retorna o nosso usuário.

2. Depois temos acesso a nossa lista de amigos, então nesse momento conseguimos acessar essas informações como retorno.

3. Para em seguida, encontrarmos esse amigo e retornar as fotos dele.

# Módulo 9 – Callback Hell (2 / 6)

Vamos aplicar o cenário anterior em um programa Javascript, simulando primeiro o comportamento de recuperação das informações do nosso usuário.

O código ao lado **simula uma operação assíncrona de recuperação de informações do usuário.** Após um atraso de 2 segundos, o callback é chamado e as informações do usuário são passadas como argumento para o console.log, que as imprime no console.

Temos uma função chamada **getUser** que recebe um callback como parâmetro. Dentro de **getUser**, usamos a função setTimeout para simular um atraso de 2 segundos antes de executar o código dentro do callback. Isso é feito para simular uma operação assíncrona que leva algum tempo para ser concluída.

Dentro do **callback do setTimeout**, temos o código console.log('Recuperando informações do usuário'), que imprime a mensagem "Recuperando informações do usuário" no console. Em seguida, chamamos o callback passado como parâmetro, passando um objeto contendo as informações do usuário: { id: 0, name: 'Daniel', surname: 'Porto' }.

No final, chamamos a função **getUser** passando console.log como o callback. Isso significa que, quando a operação assíncrona for concluída após 2 segundos, o callback será chamado e o objeto com as informações do usuário será passado como argumento para console.log, que irá imprimir esse objeto no console.

```javascript
//user <= {}
//friends => ['joão', 'bruna', 'felipe', 'Lira'];
//photos <= ['', '', '']

function getUser(callback) {
  setTimeout(() => {
    console.log('Recuperando informações do usuário');
    callback({ id: 0, name: 'Daniel', surname: 'Porto' });
  }, 2000);
}

getUser(console.log);
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL SQL CONSOLE

```
C:\Program Files\nodejs\node.exe .\..\aula_9\consoleLogging.js
Recuperando informações do usuário
> {id: 0, name: 'Daniel', surname: 'Porto'}
```

# Módulo 9 – Callback Hell (3 / 6)

Agora vamos simular o comportamento para recuperação da lista de amigos do usuário:

Nesta continuação do código, temos uma nova função chamada getFriendList que recebe dois parâmetros: user e callback. Essa função simula uma operação assíncrona de recuperação da lista de amigos do usuário.

Dentro de **getFriendList**, temos o código console.log(Recuperando lista de amigos do usuário {user.name}{user.surname}), que imprime a mensagem "Recuperando lista de amigos do usuário" seguida do nome e sobrenome do usuário no console. Em seguida, usamos a função **setTimeout** para simular um atraso de 2 segundos antes de executar o código dentro do callback. Isso é feito para simular uma operação assíncrona que leva algum tempo para ser concluída.

Dentro do **callback do setTimeout**, temos o código callback(['joão', 'bruna', 'felipe', 'Lira']), que chama o callback passado como parâmetro e passa um array contendo os nomes dos amigos do usuário: ['joão', 'bruna', 'felipe', 'Lira'].
No final, chamamos a **função getUser** passando uma função de callback como argumento. Essa função de callback recebe o objeto com as informações do usuário e chama a **função getFriendList**, passando o usuário e console.log como o callback para recuperar a lista de amigos.

```
12  function getFriendList(user, callback) {
13    console.log(
14      `Recuperando lista de amigos do usuário ${user.name} ${user.surname}`
15    );
16    setTimeout(() => {
17      callback(['joão', 'bruna', 'felipe', 'Lira']);
18    }, 2000);
19  }
20
21  getUser((user) => getFriendList(user, console.log));
22
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL SQL CONSOLE

```
C:\Program Files\nodejs\node.exe .\..\aula_9\consoleLogging.js
Recuperando informações do usuário
Recuperando lista de amigos do usuário Daniel Porto
> (4) ['joão', 'bruna', 'felipe', 'Lira']
```

Dessa forma, quando a operação assíncrona de recuperação das informações do usuário for concluída, o callback será chamado e o objeto com as informações do usuário será passado como argumento. Em seguida, a função getFriendList será chamada, exibindo a mensagem no console e, após um atraso de 2 segundos, chamando o callback novamente com a lista de amigos.

# Módulo 9 – Callback Hell (4 / 6)

E por último vamos simular o comportamento para recuperação das fotos do amigo do usuário:

```
function getPhotos(username, callback) {
  console.log(`Recuperando lista de fotos do usuário ${user.name}`);
  setTimeout(() => {
    callback(['egito', 'bahamas', 'maldivas', 'suécia']);
  }, 2000);
}

getUser((user) => getFriendList(user, (friends) => {
  getPhotos(friends[3], console.log);
}));
```

A **função getPhotos** recebe dois parâmetros: username e callback. Ela simula uma operação assíncrona de recuperação da lista de fotos do usuário.
Dentro de **getPhotos**, temos o código console.log(Recuperando lista de fotos do usuário ${user.name}), que imprime a mensagem "Recuperando lista de fotos do usuário" seguida do nome do usuário no console.

Em seguida, usamos a **função setTimeout** para simular um atraso de 2 segundos antes de executar o código dentro do callback. Isso é feito para simular uma operação assíncrona que leva algum tempo para ser concluída.

Dentro do **callback do setTimeout**, temos o código callback(['egito', 'bahamas', 'maldivas', 'suécia']), que chama o callback passado como parâmetro e passa um array contendo os nomes das fotos do usuário: ['egito', 'bahamas', 'maldivas', 'suécia'].
No final, temos a chamada encadeada de funções. Primeiro, chamamos a função getUser passando uma função de callback como argumento. Essa função de callback recebe o objeto com as informações do usuário e chama a função getFriendList, passando o usuário e uma nova função de callback como argumento.

Dentro dessa **nova função de callback**, chamamos a função getPhotos passando o quarto amigo da lista de amigos (friends[3]) e console.log como o callback para recuperar a lista de fotos.
Dessa forma, quando a operação assíncrona de recuperação das informações do usuário for concluída, o primeiro callback será chamado e o objeto com as informações do usuário será passado como argumento. Em seguida, a função getFriendList será chamada, exibindo a mensagem no console e, após um atraso de 2 segundos, chamando o segundo callback com a lista de amigos.
Por fim, **o segundo callback** chama a função getPhotos passando o quarto amigo da lista de amigos e console.log como o callback para recuperar a lista de fotos. Após o atraso de 2 segundos, o callback é chamado e a lista de fotos é passada como argumento para o console.log, que a imprime no console.

Analisando o nosso código percebemos que ele está seguindo um conceito de Callback Hell:

**CALLBACK HELL**

Callback hell, em português "inferno de callbacks", é um termo usado para descrever uma situação em que o código JavaScript se torna difícil de ler e manter devido ao aninhamento excessivo de callbacks.
Quando trabalhamos com operações assíncronas em JavaScript, como requisições AJAX, leitura de arquivos ou chamadas de API, é comum usar callbacks para lidar com o resultado dessas operações. No entanto, se várias operações assíncronas são realizadas em sequência e cada uma delas requer uma ação adicional, o código pode se tornar profundamente aninhado, dificultando a leitura e a compreensão.

No nosso programa, temos três funções assíncronas. Cada uma delas recebe um callback que será executado quando a operação assíncrona for concluída. No entanto, como essas operações são realizadas em sequência e cada uma delas depende do resultado da anterior, os callbacks são aninhados, criando uma estrutura de código confusa e difícil de acompanhar.

# Módulo 9 – Callback Hell (6 / 6)

A **callback hell** pode levar a problemas como:

- Dificuldade de leitura: O código aninhado torna difícil entender a sequência de operações e o fluxo do programa.
- Dificuldade de manutenção: Quando é necessário adicionar ou modificar uma operação assíncrona, é necessário navegar por vários níveis de aninhamento, o que pode levar a erros e tornar a manutenção do código mais complexa.
- Código propenso a erros: O aninhamento excessivo de callbacks pode levar a erros de lógica e dificultar a depuração.

Para evitar a callback hell, existem abordagens alternativas, como Promises e async/await, que fornecem uma sintaxe mais limpa e organizada para lidar com operações assíncronas em JavaScript. Essas abordagens ajudam a evitar o aninhamento excessivo de callbacks e tornam o código mais legível e fácil de manter.

**Promises**

As **Promises** em JavaScript são um recurso que permite lidar com operações assíncronas de forma mais eficiente e organizada. Elas são usadas para tratar tarefas que podem levar algum tempo para serem concluídas, como fazer uma requisição HTTP, ler um arquivo ou consultar um banco de dados.

Uma **Promise** é um objeto que representa o resultado futuro de uma operação assíncrona. Ela pode estar em um dos três estados: pendente, realizada ou rejeitada. Quando uma Promise é criada, ela é iniciada no estado pendente. À medida que a operação assíncrona é executada, a Promise pode ser resolvida com um valor ou rejeitada com um motivo de erro.

As **Promises** em JavaScript podem estar em três estados diferentes: pendente, realizada ou rejeitada. Esses estados representam o progresso da Promise e indicam em qual etapa ela se encontra.

É importante ressaltar que uma vez que uma Promise entra nos estados realizada ou rejeitada, ela se torna imutável, ou seja, não pode mais mudar de estado. Isso garante que o resultado da operação assíncrona seja tratado corretamente e que os **métodos then ou catch** sejam chamados de acordo com o resultado final da Promise.

Os **estados das Promises** em JavaScript permitem um controle mais preciso e estruturado do fluxo assíncrono, facilitando o tratamento de resultados e erros de operações que levam tempo para serem concluídas.

**Estados Promises**

- **Estado Pendente:** Quando uma Promise é criada, ela inicia no estado pendente. Isso significa que a operação assíncrona associada à Promise ainda está em andamento e aguardando a conclusão. Nesse estado, a Promise pode eventualmente ser resolvida com um valor ou rejeitada com um motivo de erro.

- **Estado Realizada:** Uma Promise entra no estado realizada quando a operação assíncrona é concluída com sucesso. Isso significa que a Promise foi resolvida com um valor resultante. Quando uma Promise está no estado realizada, ela chama a função resolve passando o valor resultante como argumento. Os métodos then encadeados à Promise serão executados, recebendo o valor resolvido como parâmetro.

- **Estado Rejeitada:** Uma Promise entra no estado rejeitada quando ocorre um erro durante a operação assíncrona. Isso significa que a Promise não foi capaz de cumprir sua promessa e foi rejeitada com um motivo de erro. Quando uma Promise está no estado rejeitada, ela chama a função reject passando o motivo do erro como argumento. Os métodos catch encadeados à Promise serão executados, recebendo o motivo do erro como parâmetro.

# Módulo 9 – Promises (3 / 5)

As Promises em JavaScript são uma funcionalidade nativa da linguagem e são implementadas pelo próprio ambiente de execução JavaScript, como o navegador web ou o ambiente Node.js. Portanto, o responsável pela implementação das Promises em JavaScript é o ambiente de execução em que o código JavaScript está sendo executado.

No caso dos navegadores web, a implementação das Promises é fornecida pelo mecanismo JavaScript do navegador, como o V8 no Google Chrome ou o SpiderMonkey no Firefox. Esses mecanismos JavaScript são responsáveis por interpretar e executar o código JavaScript, incluindo a execução das Promises.

No ambiente Node.js, a implementação das Promises é fornecida pelo próprio mecanismo JavaScript do Node.js, que é baseado no mecanismo V8. O Node.js também fornece um módulo chamado "util" que contém uma implementação adicional de Promises chamada "promisify", que permite converter funções assíncronas baseadas em callbacks em Promises.

# Módulo 9 – Promises (4 / 5)

Para utilizar Promises em JavaScript, você pode criar uma nova Promise usando a **classe Promise**. A Promise recebe uma função executora que contém a lógica da operação assíncrona.
Essa função executora recebe dois parâmetros: **resolve e reject**. O resolve é uma função que deve ser chamada quando a operação é concluída com sucesso, passando o valor resultante como argumento. O reject é uma função que deve ser chamada quando ocorre um erro, passando o motivo do erro como argumento.

Depois de criar uma Promise, você pode encadear **métodos como then e catch** para lidar com o resultado da Promise. O método then é chamado quando a Promise é resolvida com sucesso e recebe o valor resultante como argumento. O método catch é chamado quando a Promise é rejeitada e recebe o motivo do erro como argumento.

Por exemplo, vamos supor que você queira fazer uma requisição HTTP para obter dados de um servidor. Você pode criar uma Promise para essa operação e encadear métodos then e catch para lidar com o resultado:

```
const promise = new Promise((resolve, reject) => {
  // Lógica da operação assíncrona, como fazer uma requisição HTTP
  // Se a requisição for bem-sucedida, chame resolve com os dados
  // Se ocorrer um erro, chame reject com o motivo do erro
});

promise.then((data) => {
  // Manipule os dados obtidos da requisição
}).catch((error) => {
  // Manipule o erro ocorrido na requisição
});
```

# Módulo 9 – Promises (5 / 5)

As Promises em JavaScript são uma forma mais elegante e legível de lidar com operações assíncronas, evitando o chamado "callback hell" e facilitando o tratamento de erros. Elas são amplamente utilizadas em bibliotecas e frameworks modernos, como o Node.js e o React.

**Mais um exemplo aplicando os conceitos de promises e seus métodos then e catch**:

promises > promise.js > ...

```
1   const fluminenseCampeaoLibertadores2023 = true;
2
3   const promessa = new Promise((resolve, reject) => {
4     setTimeout(() => {
5       if (fluminenseCampeaoLibertadores2023) {
6         resolve('Toma aqui seu pagamento! Exatamente como combinamos! Nense!');
7       } else {
8         reject(
9           'Não acredito que ele perdeu! Infelizmente não vou conseguir te pagar!'
10        );
11      }
12    }, 2000);
13  });
14    promessa
15      .then(response => {
16        console.log(response);
17        console.log('Pooo, nem acredito que você me pagou!');
18      })
19      .catch(error => console.error(error));
20
```

Nesse momento vamos refatorar o código que criamos na aula Callback Hell utilizando a Promise e seus métodos.

original.js > getFriendList

```
1  function getUser() {
2    console.log('Recuperando informações do usuário');
3    const promise1 = new Promise(resolve => {
4      setTimeout(() => {
5        resolve({ id: 0, name: 'Daniel', surname: 'Porto' });
6      }, 2000);
7    });
8    return promise1;
9  }
```

A função getUser() é uma função que retorna uma Promise. Uma Promise é um objeto que representa a eventual conclusão (ou falha) de uma operação assíncrona e seu valor resultante.

Dentro da função getUser(), há um console.log() que imprime a mensagem "Recuperando informações do usuário". Em seguida, é criada uma nova Promise chamada promise1. Essa Promise é resolvida após um atraso de 2000 milissegundos (ou 2 segundos) usando a função setTimeout().

Dentro do setTimeout(), a Promise é resolvida com um objeto contendo as informações do usuário, como o id, nome e sobrenome.

Por fim, a função getUser() retorna a Promise promise1.

Aplicaremos os mesmos conceitos nas próximas duas funções que recupera a lista de amigos e as fotos. Então não utilizaremos mais as callbacks e sim criaremos promises e retornaremos a resolução delas.

```
function getFriendList(user) {
  console.log(
    `Recuperando lista de amigos do usuário ${user.name} ${user.surname}`
  );
  const promise2 = new Promise(resolve => {
    setTimeout(() => {
      resolve(['joão', 'bruna', 'felipe', 'Lira']);
    }, 2000);
  });
  return promise2;
}
```

```
function getPhotos(username) {
  console.log(`Recuperando fotos do usuário ${username}`);
  const promise3 = new Promise(resolve => {
    setTimeout(() => {
      resolve(['egito', 'bahamas', 'maldivas', 'suécia']);
    }, 2000);
  });
  return promise3;
}
```

E por fim, refatoramos a execução das nossas funções, para que nosso programa funcione de forma mais legível, sem criar situações de callbackhell.

```
getUser()
  .then(user => getFriendList(user))
  .then(friends => getPhotos(friends[3]))
  .then(photos => console.log(photos));
```

```
/Users/danielporto/.nvm/versions/node/v15.13.0/bin/node ./original.js
Recuperando informações do usuário
Recuperando lista de amigos do usuário Daniel Porto
Recuperando fotos do usuário Lira
> (4) ['egito', 'bahamas', 'maldivas', 'suécia']
```

A sequência de código getUser().then(user => getFriendList(user)).then(friends => getPhotos(friends[3])).then(photos => console.log(photos)) é uma cadeia de chamadas de função usando o método then() das Promises.
A função getUser() é chamada primeiro. Ela retorna uma Promise que, quando resolvida, fornece informações sobre um usuário. Em seguida, o método then() é usado para encadear a próxima chamada de função getFriendList(user), passando o usuário retornado como argumento. Essa função retorna uma Promise que, quando resolvida, fornece uma lista de amigos do usuário.
Em seguida, o método then() é usado novamente para encadear a próxima chamada de função getPhotos(friends[3]). Nesse caso, o elemento de índice 3 da lista de amigos é passado como argumento. Essa função retorna uma Promise que, quando resolvida, fornece as fotos relacionadas a esse amigo.
Por fim, o método then() é usado mais uma vez para encadear a última chamada de função console.log(photos). Quando a Promise anterior for resolvida, o resultado (as fotos) será passado como argumento para essa função, que irá imprimir as fotos no console.
Essa sequência de chamadas de função usando o método then() permite que as Promises sejam executadas em ordem, aguardando a resolução de cada uma antes de prosseguir para a próxima. Isso garante que as operações assíncronas sejam executadas de forma sincronizada e que os resultados sejam tratados corretamente.

# Módulo 9 – Tratando erros em Promises (1 / 2)

Na aula passada, refatoramos o código com o objetivo de resolver o problema das "callbacks hell" e utilizar o conceito de promessas (promises). No entanto, até agora só utilizamos as promessas quando elas são resolvidas com sucesso. Nesta aula, iremos adicionar a lógica para lidar com erros no nosso programa e quando a promessa for rejeitada.

Vamos imaginar o cenário no qual não conseguimos recuperar as fotos, pois o perfil do usuário (amigo) está desabilitado.

```
function getPhotos(username) {
  console.log(`Recuperando fotos do usuário ${username}`);
  const promise3 = new Promise((resolve, reject) => {
    setTimeout(() => {
      reject('Não foi possível acessar as fotos. Usuário desabitou o perfil');
    }, 2000);
  });
  return promise3;
}
```

No código acima estamos dizendo que a nossa promisse será rejeitada e precisamos fazer um tratamento de erro para ela, e também para possíveis promises que venham a ser rejeitadas futuramente. Então para isso usaremos os métodos catch e finally.

# Módulo 9 – Tratando erros em Promises (2 / 2)

```
39  getUser()
40    .then(user => getFriendList(user))
41    .then(friends => getPhotos(friends[3]))
42    .then(photos => console.log(photos))
43    .catch(error => console.error(error))
44    .finally(() => {
45      console.log('Esse é o fim da comunicação com o instagram');
46    });
47
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
/Users/danielporto/.nvm/versions/node/v15.13.0/bin/node ./promisified.js
Recuperando informações do usuário
Recuperando lista de amigos do usuário Daniel Porto
Recuperando fotos do usuário Lira
Não foi possível acessar as fotos. Usuário desabitou o perfil
Esse é o fim da comunicação com o instagram
```

No código ao lado, podemos observar que utilizamos a estrutura "catch" para lidar com o erro que recebemos da nossa promessa rejeitada. Ao fazer esse tratamento, evitamos que ocorra um erro no nosso terminal.

Além disso, utilizamos o "finally" para garantir que um bloco de instruções seja executado, independentemente de a promessa ter sido resolvida com sucesso ou rejeitada. No nosso caso, estamos utilizando o "finally" para indicar o encerramento da comunicação.

# Módulo 9 – Async / Await (1 / 3)

O **"async/await"** é uma funcionalidade do JavaScript que permite escrever código assíncrono de forma mais síncrona e legível. Ele é baseado no **uso de promessas (promises)** e fornece uma maneira mais simples 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, sem a necessidade de encadeamento de callbacks ou uso explícito de métodos "then" e "catch". Isso torna o código mais legível e mais fácil de entender.

Aqui está um exemplo de como usar o **"async/await"** em JavaScript:

```javascript
async function minhaFuncaoAssincrona() {
  try {
    const resultado = await minhaPromessa(); // Aguarda a resolução da promessa
    console.log(resultado); // Executa após a promessa ser resolvida
  } catch (erro) {
    console.log(erro); // Trata o erro caso a promessa seja rejeitada
  } finally {
    console.log("Fim da execução"); // Executa sempre, independentemente do resultado da promessa
  }
}

minhaFuncaoAssincrona();
```

# Módulo 9 – Async / Await (2 / 3)

No exemplo anterior, a função "minhaFuncaoAssincrona" é declarada como assíncrona usando a palavra-chave "async". Dentro dela, usamos o "await" antes da chamada da função "minhaPromessa", que retorna uma promessa. O código após o "await" só será executado quando a promessa for resolvida.

No bloco "try", tratamos o resultado da promessa resolvida. Se ocorrer algum erro e a promessa for rejeitada, o bloco "catch" será executado para tratar o erro.

Por fim, o bloco "finally" é opcional e será executado sempre, independentemente do resultado da promessa. É útil para realizar ações de encerramento ou limpeza.
Dessa forma, o "async/await" simplifica a escrita e o tratamento de código assíncrono em JavaScript, tornando-o mais legível e fácil de manter.

Agora vamos aplicar esse conceito na estrutura de código da aula passada, vamos tratar as promises utilizando o async / await.

```
async function displayLirasPhotos() {
  const user = await getUser();
  const friends = await getFriendList(user);
  const photos = await getPhotos(friends[3]);
  console.log(photos);
}

displayLirasPhotos();
```

```
PROBLEMS    OUTPUT    TERMINAL    DEBUG CONSOLE
 /Users/danielporto/.nvm/versions/node/v15.13.0/bin/node ./promisified.js
 Recuperando informações do usuário
 Recuperando lista de amigos do usuário Daniel Porto
 Recuperando fotos do usuário Lira
> (4) ['egito', 'bahamas', 'maldivas', 'suécia']
```

# Módulo 9 – Async / Await (3 / 3)

E como vimos anteriormente, conseguimos utilizar try / catch / finally para realizarmos o tratamento de erro dentro da nossa função async / await:

```
async function displayLirasPhotos() {
  try {
    const user = await getUser();
    const friends = await getFriendList(user);
    const photos = await getPhotos(friends[3]);
    console.log(photos);
  } catch (error) {
    console.error(error);
  } finally {
    console.log('Terminando a conexão com o instagram');
  }
}

displayLirasPhotos();
```

# Módulo 9 – Múltiplas Promises (1 / 4)

Em JavaScript, é comum trabalhar com múltiplas promessas ao lidar com operações assíncronas. As múltiplas promessas podem ser executadas em paralelo ou em sequência, dependendo dos requisitos do seu código.
Uma abordagem para lidar com múltiplas promessas em JavaScript é o Promise.all:

**Promise.all**

A função Promise.all é uma funcionalidade do JavaScript que permite lidar com múltiplas promessas de forma conveniente. Ela recebe um array de promessas como argumento e retorna uma nova promessa. Essa nova promessa será resolvida somente quando todas as promessas do array forem resolvidas, ou será rejeitada se qualquer uma das promessas for rejeitada.

Aqui está uma explicação detalhada de como a função Promise.all funciona:

- Recebendo um array de promessas: A função Promise.all recebe um array contendo uma ou mais promessas. Cada promessa representa uma operação assíncrona que será executada em paralelo.
- Retornando uma nova promessa: A função Promise.all retorna uma nova promessa que será resolvida quando todas as promessas do array forem resolvidas. Essa nova promessa representa o resultado combinado de todas as promessas.
- Resolução das promessas: A nova promessa será resolvida com um array contendo os resultados das promessas originais, na mesma ordem em que foram passadas para Promise.all. Os resultados são obtidos quando todas as promessas forem resolvidas com sucesso.
- Rejeição das promessas: Se qualquer uma das promessas do array for rejeitada, a nova promessa retornada por Promise.all será imediatamente rejeitada. O motivo da rejeição será o mesmo da primeira promessa que foi rejeitada.
- Tratamento de erros: Para tratar erros, podemos encadear um método catch na nova promessa retornada por Promise.all. Esse método será chamado se alguma das promessas for rejeitada.

# Módulo 9 – Múltiplas Promises (2 / 4)

Vamos criar um exemplo para observarmos o comportamento do Promise.all:

```
multiplePromises.js > ...
const promise1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve(1500);
  }, 2000);
});

const promise2 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve(3500);
  }, 2000);
});

Promise.all([promise1, promise2]).then(console.log);
```

```
/Users/danielporto/.nvm/versions/node/v15.13.0/bin/node ./multiplePromises.js
> (2) [1500, 3500]
```

Neste exemplo, temos duas promessas, promise1 e promise2, que são resolvidas após um determinado tempo. Ao passar essas promessas para Promise.all, uma nova promessa é retornada.

Quando ambas as promessas forem resolvidas, o método then será chamado, recebendo e retornando um console.log com o resultado.

# Módulo 9 – Múltiplas Promises (3 / 4)

Vamos observar o comportamento e o tempo que as nossas promises retornam quando utilizamos o promise.all:

```
15  async function generateTotalPrice() {
16    const startTime = Date.now();
17    Promise.all([
18      new Promise((resolve, reject) => {
19        setTimeout(() => {
20          resolve(1500);
21        }, 2000);
22      }),
23      new Promise((resolve, reject) => {
24        setTimeout(() => {
25          resolve(3500);
26        }, 2000);
27      }),
28    ]);
29    const endTime = Date.now();
30    console.log(`elapsedTime ${endTime - startTime}ms`);
31    console.log([price1, price2]);
32  }
33
34  generateTotalPrice();
35
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
/Users/danielporto/.nvm/versions/node/v15.13.0/bin/node ./multiplePromises.js
elapsedTime 4002ms
> (2) [1500, 3500]
```

# Módulo 9 – Múltiplas Promises (4 / 4)

Vamos verificar um outro comportamento do nosso código, alterando o nosso código e nossas promises.

multiplePromises.js > generateTotalPrice > <function> > setTimeout() callback

```
14
15  async function generateTotalPrice() {
16    const startTime = Date.now();
17    Promise.all([
18      new Promise((resolve, reject) => {
19        setTimeout(() => {
20          resolve(1500);
21        }, 2000);
22      }),
23      new Promise((resolve, reject) => {
24        setTimeout(() => {
25          resolve(3500);
26        }, 2000);
27      }),
28    ])
29      .then(console.log)
30      .then(() => {
31        const endTime = Date.now();
32        console.log(`elapsedTime ${endTime - startTime}ms`);
33      });
34  }
35
36  generateTotalPrice();
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
/Users/danielporto/.nvm/versions/node/v15.13.0/bin/node ./multiplePromises.js
> (2) [1500, 3500]
elapsedTime 2005ms
```

Podemos perceber a diferença no tempo de execução entre os dois programas. Isso acontece porque, no exemplo 1, temos duas promessas criadas separadamente, enquanto no exemplo 2, as promessas são criadas dentro do Promise.all. Isso significa que o tempo de execução e espera pelo retorno das promessas será diferente.

No entanto, há um problema na abordagem do exemplo 2. Se uma das promessas gerar um erro, o programa será executado com erro e não será capaz de resolver a promessa que teve um retorno bem-sucedido.
Isso ocorre porque o Promise.all trata as promessas como um conjunto, ou seja, se uma delas falhar, todas as outras também serão consideradas falhas. Isso pode ser um problema se quisermos lidar com erros individualmente ou se quisermos obter os resultados das promessas que foram resolvidas com sucesso, mesmo que outras tenham falhado.

Para resolver esse problema, podemos utilizar o tratamento de erros individualmente para cada promessa, em vez de usar o Promise.all. Dessa forma, podemos lidar com os erros de forma mais granular e obter os resultados das promessas que foram resolvidas com sucesso, mesmo que outras tenham falhado.

Módulo 9

# OBJETOS

OBJETOS

OBJETOS

# Módulo 10 – Criando um Objeto: Object Literal (1 / 3)

Uma das características mais poderosas e versáteis do JavaScript é sua capacidade de trabalhar com objetos.
Os **objetos** são como os blocos de construção fundamentais do JavaScript, permitindo que você organize e manipule dados de maneira eficiente e estruturada. Eles são a espinha dorsal de muitas aplicações web complexas, permitindo que você crie estruturas de dados personalizadas, modele entidades do mundo real e até mesmo construa classes e instâncias para criar código mais reutilizável.

Neste módulo, começaremos explorando a estrutura inicial de um objeto, conhecida como **'object literals'** ou literais de objeto em JavaScript. Você aprenderá a criar objetos de forma simples e direta, definindo propriedades e valores iniciais. Além disso, mergulharemos fundo no conceito de objetos em JavaScript, explorando sua sintaxe, propriedades, métodos e como eles podem ser usados para resolver problemas do mundo real.

Ao longo deste módulo, você desenvolverá uma compreensão sólida e prática de como criar, manipular e interagir com objetos JavaScript, capacitando você a escrever código mais eficiente, limpo e poderoso. Vamos começar explorando os fundamentos dos literais de objeto e, em seguida, avançaremos para tópicos mais avançados para aprofundar ainda mais seu conhecimento sobre objetos em JavaScript.

# Módulo 10 – Criando um Objeto: Object Literal (2 / 3)

Para criar um objeto em JavaScript, você pode usar várias abordagens, mas uma das maneiras mais simples e comuns é usando a sintaxe de literais de objeto, que é como um par de chaves **{}**. Vamos criar um objeto chamado **account1** usando a sintaxe de objeto literal, que representa uma conta de um banco.

```
JS objectLiteral.js > ...
1   //object literal =
2   const account1 = {
3
4   }
```

Agora vamos construir as propriedades que o nosso objeto irá armazenar e os valores de cada propriedade:

- **name: "Daniel Porto"**: Define a propriedade **name** com o valor **"Daniel Porto"**. Esta propriedade armazena o nome do titular da conta.
- **accountNumber: "12345678-9"**: Define a propriedade **accountNumber** com o valor **"12345678-9"**. Esta propriedade armazena o número da conta.
- **balance: 500**: Define a propriedade **balance** com o valor **500**. Esta propriedade armazena o saldo da conta, que é definido como **500**.
- **displayBalance: function () { ... }**: Define a propriedade **displayBalance** como uma função anônima. Esta função é responsável por exibir o saldo da conta usando **console.log(this.balance)**. A palavra-chave **this** refere-se ao objeto atual (**account1** neste caso), permitindo que a função acesse a propriedade **balance** do objeto.

# Módulo 10 – Criando um Objeto: Object Literal (3 / 3)

Portanto, **account1** é um objeto que representa uma conta bancária com três propriedades (nome, número da conta e saldo) e um método (displayBalance) para exibir o saldo.

```
JS objectLiteral.js > ...
1  //object literal
2  const account1 = {
3    name: 'Daniel Porto',
4    accountNumber: '12345678-9',
5    balance: 500,
6    displayBalance: function () {
7      console.log(this.balance);
8    },
9  };
```

Para exibir nosso objeto no console, utilizaremos a função **console.log**, e também aprimoraremos a propriedade **displayBalance** para demonstrar maior complexidade. Vamos empregar string template para combinar texto com nossas variáveis de forma mais dinâmica e expressiva.

```
JS objectLiteral.js > ...
1  //object literal
2  const account1 = {
3    name: 'Daniel Porto',
4    accountNumber: '12345678-9',
5    balance: 500,
6    displayBalance: function () {
7      console.log(
8        `account number ${this.accountNumber} has a total balance of ${this.balance}`
9      );
10   },
11 };
12
13 console.log(account1);
14 account1.displayBalance();
```

Vamos executar o nosso programa no terminal do VS Code, com o comando node ./nomePrograma.js e podemos analisar o retorno do objeto criado e do método displayBalance.

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   SQL CONSOLE
  C:\Program Files\nodejs\node.exe .\objectLiteral.js
> {name: 'Daniel Porto', accountNumber: '12345678-9', balance: 500, displayBalance: ƒ}
  account number 12345678-9 has a total balance of 500
```

# Módulo 10 – Criando um Objeto: Função Fábrica (1 / 3)

Utilizar **object literals** para criar objetos com a mesma estrutura não é uma abordagem eficiente. Isso resulta em redundância de código, dificulta a manutenção à medida que o número de objetos aumenta e não favorece a reutilização de código. Além disso, essa abordagem não é escalável quando se trata de alterar a estrutura do objeto ou adicionar métodos compartilhados.

Uma alternativa recomendada é adotar **funções fábrica**, que permitem a criação de objetos com a mesma estrutura de maneira mais eficaz, promovendo a reutilização, a facilidade de manutenção e a expansão dos objetos.

Uma **função fábrica de objetos** é uma função em JavaScript projetada para **criar e retornar novos objetos com uma estrutura específica.** Em vez de criar objetos manualmente ou replicar literais de objeto, você encapsula a lógica de criação de objetos com propriedades e valores predefinidos em uma função. Isso torna o processo de criação de objetos mais eficiente e facilita a manutenção.

factoryFunction.js > createAccount

```js
//Função Fábrica - Factory Function

function createAccount(username, accountNumber, initialBalance) {
  return {
    name: username,
    accountNumber: accountNumber,
    balance: initialBalance,
    displayBalance: function () {
      console.log(
        `account number ${this.accountNumber} has a total balance of ${this.balance}`
      );
    },
  };
}
```

# Módulo 10 – Criando um Objeto: Função Fábrica (2 / 3)

A função **createAccount** é uma função fábrica que cria e retorna objetos representando contas bancárias. Ela aceita três parâmetros:

1. **username**: Este parâmetro representa o nome do titular da conta.
2. **accountNumber**: Este parâmetro representa o número da conta.
3. **initialBalance**: Este parâmetro representa o saldo inicial da conta.

Dentro da função **createAccount**, um objeto é criado usando a sintaxe de objeto literal. Este objeto tem quatro propriedades:

- **name**: Armazena o nome do titular da conta, que é definido com base no valor do parâmetro **username**.
- **accountNumber**: Armazena o número da conta, que é definido com base no valor do parâmetro **accountNumber**.
- **balance**: Armazena o saldo inicial da conta, que é definido com base no valor do parâmetro **initialBalance**.
- **displayBalance**: É uma função anônima que pertence ao objeto. Quando chamada, essa função exibe no console uma mensagem que inclui o número da conta e o saldo da conta.

O uso da função fábrica **createAccount** permite criar objetos de conta com facilidade e consistência, fornecendo valores iniciais para o nome, número da conta e saldo. Em seguida, o método **displayBalance** pode ser chamado em um objeto de conta para exibir informações sobre o saldo da conta.

# Módulo 10 – Criando um Objeto: Função Fábrica (3 / 3)

Criada a funcionalidade da função fábrica createAccount, podemos utilizar para inicializar um objeto que representa uma conta bancária.

```
const account1 = createAccount('Daniel Porto', '12345678-9', 500);
console.log(account1);
account1.displayBalance();
```

Após a execução dessa linha de código, **account1** é um objeto que representa uma conta bancária com o nome "Daniel Porto", número de conta "12345678-9" e saldo inicial de 500. Você pode acessar e manipular essas propriedades chamando **account1.name**, **account1.accountNumber** e **account1.balance**, e também pode usar o método **account1.displayBalance()** para exibir o saldo da conta no console.

Para criar um novo objeto, vamos utilizar novamente a função fábrica createAccount:

```
const account2 = createAccount('João Lira', '98765432-1', 50000);
console.log(account2);
account2.displayBalance();
```

```
C:\Program Files\nodejs\node.exe .\factoryFunction.js
> {name: 'Daniel Porto', accountNumber: '12345678-9', balance: 500, displayBalance: ƒ}
account number 12345678-9 has a total balance of 500
> {name: 'João Lira', accountNumber: '98765432-1', balance: 50000, displayBalance: ƒ}
account number 98765432-1 has a total balance of 50000
```

# Módulo 10 – Criando um Objeto: Construtor (1 / 3)

Vamos conhecer sobre o conceito **Construtor**. Um construtor em JavaScript é uma função especial que desempenha o papel de um modelo ou fábrica para criar e inicializar objetos com estrutura predefinida. A característica distintiva de um construtor é o uso da palavra-chave **this** dentro da função, que se refere ao objeto sendo criado ou inicializado. Dentro do construtor, você pode definir propriedades e métodos que serão compartilhados por todas as instâncias de objetos criados a partir dele.

Nesse contexto, criaremos a entidade **Account** que representa uma conta bancária por meio de um construtor personalizado chamado **Account**. Este construtor aceita parâmetros como nome do titular, número da conta e saldo inicial e, em seguida, inicializa as propriedades desses objetos com base nos valores fornecidos. Assim, cada vez que você usa o construtor **Account** para criar uma nova conta, você está criando uma instância de objeto que compartilha a mesma estrutura e os mesmos métodos definidos no construtor, tornando a criação e o gerenciamento de objetos mais organizados e eficientes.

```
// PascalCase (pascal case)

function Account(username, accountNumber, initialBalance) {
  this.name = username;
  this.accountNumber = accountNumber;
  this.balance = initialBalance;
  this.displayBalance = function () {
    console.log(
      `account number ${this.accountNumber} has a total balance of ${this.balance}`
    );
  };
}
```

# Módulo 10 – Criando um Objeto: Construtor (2 / 3)

A convenção de nomeação de funções construtoras em JavaScript usando a primeira letra maiúscula não segue a regra de camelCase por um motivo específico: diferenciação. A convenção, **pascalCase**, de usar uma letra maiúscula no início do nome de uma função construtora torna mais fácil identificar e distinguir construtores de outras funções e variáveis em seu código.

Quando você olha para o código JavaScript, a primeira letra maiúscula no nome de uma função construtora indica que ela deve ser usada com o operador **new** para criar objetos. Isso ajuda a evitar erros, pois cria uma clara distinção entre funções regulares e funções construtoras. Além disso, ajuda na legibilidade do código, tornando mais evidente quando você está criando instâncias de objetos.

```
const account1 = new Account('Daniel Porto', '12345678-9', 500);
console.log(account1);
account1.displayBalance();

const account2 = new Account('João Lira', '98765432-1', 50000);
console.log(account2);
account2.displayBalance();
```

# Módulo 10 – Criando um Objeto: Construtor (3 / 3)

A principal vantagem das funções construtoras em relação às funções fábrica é a capacidade de criar objetos que compartilham um protótipo comum.

Isso permite a economia de memória e a eficiência ao compartilhar métodos e propriedades entre as instâncias. Além disso, o uso de funções construtoras segue uma convenção de nomenclatura, tornando a identificação de instâncias mais clara com a primeira letra maiúscula. O operador **new** também é usado, aumentando a legibilidade.

No entanto, a escolha entre construtoras e fábricas depende dos requisitos do projeto e do estilo de programação preferido, e ambas podem ser usadas em conjunto, se necessário.

```
  C:\Program Files\nodejs\node.exe .\constructor.js
> Account {name: 'Daniel Porto', accountNumber: '12345678-9', balance: 500, displayBalance: ƒ}
  account number 12345678-9 has a total balance of 500
> Account {name: 'João Lira', accountNumber: '98765432-1', balance: 50000, displayBalance: ƒ}
  account number 98765432-1 has a total balance of 50000
```

# Módulo 10 – [Opcional] Conhecendo as partes do Computador (1 / 2)

Um computador é composto por várias peças fundamentais que desempenham papéis cruciais em seu funcionamento, vou focar a explicação nas três principais peças do computador: CPU, Memória RAM e Disco Rígido/ HD (ou SSD):

**CPU (Unidade Central de Processamento):** A CPU é o equivalente ao processador do JavaScript. Ela é responsável por executar todas as instruções e operações dentro de um programa JavaScript. Assim como a CPU controla o funcionamento do computador, a CPU do JavaScript interpreta e executa o código JavaScript, realizando cálculos, operações lógicas e gerenciando o fluxo de controle.

**Memória RAM (Memória de Acesso Aleatório):** A Memória RAM é como a área de trabalho temporária do JavaScript. Quando você executa um programa JavaScript em um navegador ou ambiente de desenvolvimento, a Memória RAM é usada para armazenar temporariamente variáveis, objetos e dados necessários para a execução do código. Quanto mais RAM disponível, mais eficiente será a execução de programas JavaScript complexos, especialmente aqueles que envolvem grandes conjuntos de dados.

**Disco Rígido (ou SSD - Unidade de Estado Sólido):** O Disco Rígido (ou SSD) é onde os arquivos de código JavaScript, bibliotecas e recursos da web são armazenados permanentemente. Quando você cria um site ou aplicativo da web, os arquivos JavaScript são hospedados no disco rígido do servidor da web. Quando alguém acessa o site ou aplicativo, esses arquivos são baixados para a máquina do usuário, temporariamente armazenados na RAM e, em seguida, executados pela CPU local. Isso permite que os programas JavaScript sejam executados no navegador e interajam com os usuários.

# Módulo 10 – [Opcional] Conhecendo as partes do Computador (2 / 2)

Em resumo, em programação JavaScript, a CPU executa o código, a Memória RAM armazena temporariamente os dados necessários para a execução, e o Disco Rígido (ou SSD) armazena permanentemente os arquivos JavaScript e recursos da web. Esses componentes trabalham juntos para permitir a criação e a execução de aplicativos web dinâmicos e interativos.

Vamos analisar o seguinte código e a saída que ele possui ao executá-lo:

Users > danielporto > stackexample.js > comportamentoB

```javascript
function comportamentoA() {
  console.log('estou executando o comportamento A');
  comportamentoB();
  console.log('Finalizei a execução do comportamento A');
}

function comportamentoB() {
  console.log('estou executando o comportamento B');
  comportamentoC();
  console.log('Finalizei a execução do comportamento B');
}

function comportamentoC() {
  console.log('estou executando o comportamento C');
  console.log('Finalizei a execução do comportamento C');
}

comportamentoA();
```

```
/Users/danielporto/.nvm/versions/node/v15.13.0/bin/node ./stackexample.js
estou executando o comportamento A
estou executando o comportamento B
estou executando o comportamento C
Finalizei a execução do comportamento C
Finalizei a execução do comportamento B
Finalizei a execução do comportamento A
```

O código é composto por três funções, cada uma delas contendo comandos para imprimir mensagens no terminal de acordo com sua execução.

Dentro da **função "comportamentoA"**, fazemos uma chamada para a **função "comportamentoB"**, que, ao ser executada, por sua vez, chama a **função "comportamentoC"**.

Ao examinar a saída no terminal, observamos que o primeiro console.log dentro de "comportamentoA" é executado primeiro. Em seguida, a execução entra na função "comportamentoB", onde o primeiro console.log é acionado. A partir daí, ocorre a entrada na função "comportamentoC", executando todos os console.log dessa função. Em seguida, a execução retorna à função "comportamentoB", onde o segundo console.log é executado. Finalmente, a execução volta para a função "comportamentoA", onde o segundo console.log é executado e, por fim, o programa é encerrado.

**Depuração com o Visual Studio Code (VS Code):**

- O Visual Studio Code é um ambiente de desenvolvimento integrado (IDE) popular para JavaScript e outras linguagens.
- Uma das funcionalidades essenciais do VS Code é a capacidade de depuração de código, que ajuda os desenvolvedores a identificar e corrigir erros em seus programas JavaScript.
- Ao usar o VS Code para depurar JavaScript, você pode definir pontos de interrupção (breakpoints) em seu código, inspecionar variáveis, avaliar expressões em tempo de execução e acompanhar a pilha de chamada (call stack) para entender como seu código está se comportando durante a execução.

# Módulo 10 – Stack e Heap (3 / 7)

Em JavaScript, não há uma separação direta entre os conceitos de "stack" (pilha) e "heap" (monte), como ocorre em linguagens de programação de baixo nível, como C++ ou Java. No entanto, podemos discutir os conceitos relacionados à alocação de memória e ao gerenciamento de memória em JavaScript, incluindo a "pilha de chamada" (call stack).

- Em linguagens de programação, a alocação de memória pode ocorrer em dois locais principais: na pilha (stack) e no heap (monte).
- A pilha é usada para armazenar variáveis locais e contextos de função. Geralmente, a alocação na pilha é rápida, mas limitada em tamanho e escopo. Ela segue um modelo de "último a entrar, primeiro a sair" (LIFO).
- O heap é usado para alocar objetos e dados complexos cujo tamanho não é conhecido em tempo de compilação. A alocação no heap é mais flexível, mas pode ser mais lenta e requer gerenciamento de memória adequado.

**Pilha de Chamada (Call Stack):**

- A pilha de chamada, frequentemente referida como a "stack" em JavaScript, é uma estrutura de dados fundamental que rastreia a execução das funções em um programa.
- Quando uma função é chamada, um novo contexto de execução, conhecido como "frame", é empilhado na call stack. Este contexto armazena informações sobre a função, como variáveis locais e o endereço de retorno.
- À medida que as funções concluem sua execução, seus contextos são desempilhados da pilha de chamada. Esse processo é chamado de "desempilhamento".
- A pilha de chamada garante que as funções sejam executadas na ordem correta, seguindo o princípio "último a entrar, primeiro a sair" (LIFO).

**Monte (Heap):**
- Em JavaScript, o "heap" é a área de memória onde objetos complexos, como objetos e arrays, são armazenados.
- A alocação de memória no heap é gerenciada automaticamente pelo mecanismo de coleta de lixo (garbage collector) do JavaScript. Isso significa que objetos não mais referenciados são removidos da memória para liberar espaço automaticamente.
- Ao contrário de linguagens de baixo nível, você não precisa se preocupar em alocar ou desalocar memória manualmente no heap em JavaScript.

Em resumo, em JavaScript, a "pilha de chamada" (call stack) é usada para rastrear a execução de funções e a ordem em que elas foram chamadas. O "heap" é usado para armazenar objetos e dados complexos. O gerenciamento de memória, incluindo alocação e desalocação, é tratado automaticamente pelo mecanismo de coleta de lixo do JavaScript.

Vamos analisar as **diferenças entre algumas características da Stack e Heap**:

**1. Finalidade:**
- **Stack (Pilha):** A pilha é usada principalmente para armazenar dados relacionados a funções e seus contextos de execução. Ela mantém o controle da execução do programa, incluindo informações sobre variáveis locais, endereços de retorno e o fluxo de chamadas de função.
- **Heap (Monte):** O heap é usado para alocar memória dinamicamente durante a execução do programa para objetos cujo tamanho não é conhecido em tempo de compilação. Ele é usado para armazenar objetos complexos, como objetos, arrays e estruturas de dados.

**2. Escopo e Duração:**

- **Stack (Pilha):** Os dados armazenados na pilha têm um escopo local e uma duração limitada. Eles são criados quando uma função é chamada e destruídos quando a função retorna. A pilha segue um modelo "último a entrar, primeiro a sair" (LIFO).
- **Heap (Monte):** Os dados alocados no heap têm um escopo global (ou ao menos fora do escopo de uma única função) e uma duração que pode ser mais longa. Eles permanecem na memória até que sejam explicitamente desalocados ou até que o programa termine.

**3. Velocidade de Acesso:**

- **Stack (Pilha):** O acesso aos dados na pilha é geralmente mais rápido, pois envolve operações simples de empilhar e desempilhar dados. Isso a torna eficiente para variáveis locais e contextos de função.
- **Heap (Monte):** O acesso aos dados no heap pode ser mais lento, pois requer a localização de memória livre e pode envolver operações de gerenciamento de memória mais complexas. No entanto, a alocação no heap é flexível e adequada para objetos de tamanho variável.

**4. Gerenciamento de Memória:**

- **Stack (Pilha):** A gestão de memória na pilha é automatizada e segue o modelo LIFO. A memória é automaticamente liberada quando as funções retornam, o que minimiza os riscos de vazamento de memória.
- **Heap (Monte):** A gestão de memória no heap é manual ou automatizada por um coletor de lixo (garbage collector), dependendo da linguagem. Os programadores podem alocar memória no heap, mas também são responsáveis por liberá-la quando não é mais necessária. Um gerenciamento inadequado pode resultar em vazamentos de memória.

**5. Uso Típico:**

- **Stack (Pilha):** A pilha é usada para gerenciar a execução de programas, incluindo variáveis locais, argumentos de função e o controle de fluxo de chamadas de função.
- **Heap (Monte):** O heap é usado para alocar memória para objetos complexos que precisam persistir além do escopo local, como estruturas de dados dinâmicas, objetos e arrays de tamanho variável.

Vamos alterar a nossa função comportamentoA que ao invés de chamar a função comportamentoB, chamará a si mesma (lembrando que isso é possível devido ao conceito de recursividade). E agora vamos analisar o que irá ocorrer:

```
Users > danielporto > stackexample.js > comportamentoA
1  function comportamentoA() {
2    console.log('estou executando o comportamento A');
3    comportamentoA();
4    console.log('Finalizei a execução do comportamento A');
5  }
```

Perceba que um erro ocorreu devido ao fato de nossa função ter sido chamada 9073 vezes. Se esse erro não tivesse ocorrido, a função teria continuado a ser executada indefinidamente, pois não há um mecanismo para encerrar sua execução. Ela se chama repetidamente e nunca atinge a instrução **console.log('Finalizei a execução do comportamento A')**, resultando em um loop infinito.

```
PROBLEMS   OUTPUT   TERMINAL   DEBUG CONSOLE                Filter (e.g. te
/Users/danielporto/.nvm/versions/node/v15.13.0/bin/node ./stackexample.js
9073 estou executando o comportamento A
Process exited with code 1
Uncaught RangeError RangeError: Maximum call stack size exceeded
    at value (internal/console/constructor:293:13)
    at log (internal/console/constructor:363:26)
    at comportamentoA (/Users/danielporto/stackexample.js:2:11)
    at comportamentoA (/Users/danielporto/stackexample.js:3:3)
```

O **erro "Maximum Call Stack Size Exceeded" (Máximo de Tamanho da Pilha de Chamadas Excedido)** é uma mensagem de erro que ocorre em linguagens de programação, incluindo JavaScript, quando uma função faz chamadas recursivas de maneira excessiva, levando à exaustão da pilha de chamadas (stack).
Aqui está uma explicação mais detalhada do erro:

1. **Pilha de Chamadas (Call Stack):** A pilha de chamadas (call stack) é uma estrutura de dados usada para rastrear as chamadas de função em um programa. Cada vez que uma função é chamada, um novo "quadro de chamada" (call frame) é empilhado na pilha de chamadas, contendo informações sobre a função em execução, como variáveis locais e o ponto de retorno.
2. **Recursão:** A recursão é uma técnica em programação em que uma função chama a si mesma. Isso pode ser útil para resolver problemas que podem ser quebrados em subproblemas semelhantes. No entanto, é importante ter um caso base que termine a recursão, caso contrário, a chamada recursiva continuará indefinidamente.
3. **Erro "Maximum Call Stack Size Exceeded":** O erro "Maximum Call Stack Size Exceeded" ocorre quando uma função recursiva não tem um caso base adequado ou quando o número de chamadas recursivas é tão grande que a pilha de chamadas fica sem espaço para acomodar novos quadros de chamada. Isso é um erro crítico, porque significa que o programa não pode continuar a executar corretamente.

# Módulo 10 – Passagem por valor Vs. Passagem por referência (1 / 4)

Para iniciarmos essa aula, primeiro vamos criar uma função **sum(a, b)** que recebe dois números como entrada, adiciona-os e retorna o resultado da adição. A variável **a** é reatribuída com o valor da soma, mas a variável **b** permanece inalterada.

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

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

let parcel1 = 2;
let parcel2 = 3;

console.log(sum(parcel1,parcel2));
```

As variáveis **parcel1** e **parcel2** são inicializadas com os valores 2 e 3, respectivamente. Em seguida, a função **sum(parcel1, parcel2)** é chamada, onde **parcel1** e **parcel2** são passados como argumentos. O resultado da função será a soma desses dois valores, que será impresso no console. Portanto, o código imprimirá **5** no console.

```
PROBLEMS    OUTPUT    TERMINAL    DEBUG CONSOLE                    Filter (e
/Users/danielporto/.nvm/versions/node/v15.13.0/bin/node ./passingValueVsReference.js
5
```

Se você executar **console.log(parcel1)** após a chamada da função **sum(parcel1, parcel2)**, o valor de **parcel1** não será afetado pela função **sum**. Portanto, ele ainda conterá o valor original, que é **2**. O código imprimirá **2** no console.

```
/Users/danielporto/.nvm/versions/node/v15.13.0/bin/node ./passingValueVsReference.js
5
2
```

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

let parcel1 = 2;
let parcel2 = 3;

console.log(sum(parcel1,parcel2));
console.log(parcel1);
```

Isso ocorre porque a função **sum** não modifica a variável **parcel1** diretamente. Ela recebe os valores **parcel1** e **parcel2** como argumentos, realiza a adição e retorna o resultado, mas não altera a variável original. A função opera com cópias locais dos valores passados como argumentos. Portanto, qualquer modificação dentro da função não afeta as variáveis originais fora dela. Para atualizar a variável **parcel1**, você precisaria atribuir o resultado da função de volta a **parcel1**, como em **parcel1 = sum(parcel1, parcel2)**.

Em resumo, a função opera com cópias locais dos valores passados como argumentos, seguindo o **conceito de "passagem por valor**". Em linguagens como JavaScript, os valores são passados por valor, o que significa que as variáveis na função recebem cópias dos valores originais, e quaisquer modificações feitas dentro da função não afetam as variáveis fora dela.

# Módulo 10 – Passagem por valor Vs. Passagem por referência (3 / 4)

Agora vamos criar uma segunda função, **sum2** , que recebe dois objetos, **obj1** e **obj2**, como argumentos. Ela adiciona os valores das propriedades **value** desses objetos e retorna o resultado em um novo objeto. Portanto, ao chamar **sum2(obj1, obj2)**, o resultado será um novo objeto contendo o valor da soma dos valores das propriedades **value** de **obj1** e **obj2**.

```
/Users/danielporto/.nvm/versions/node/v15.13.0/bin/node ./passingValueVsReference.js
5
2
{value: 5}
```

```
function sum2(obj1, obj2) {
    obj1.value = obj1.value + obj2.value;
    return obj1;
}

let obj1 = {
    value: 2
};

let obj2 = {
    value: 3
};

console.log(sum(obj1,obj2));
```

Nesse momento, ao imprimir o obj1 com **console.log**, você verá o valor de **obj1** após a chamada da função **sum2**. Como os objetos em JavaScript são passados por referência, a função **sum2** modificou diretamente o objeto **obj1**. Portanto, o segundo **console.log** mostrará **obj1** com o valor modificado **{ value: 5 }**, refletindo a alteração feita pela função **sum2**, já que **obj1** é passado por referência.

```
/Users/danielporto/.nvm/versions/node/v15.13.0/bin/node ./passingValueVsReference.js
5
2
5
{value: 5}
```

```
console.log(sum(obj1,obj2));
console.log(obj1);
```

# Módulo 10 – Passagem por valor Vs. Passagem por referência (4 / 4)

A diferença entre passagem por valor e referência também está relacionada à forma como a memória é gerenciada no heap e na stack em JavaScript.

- Na função **sum**, que opera com números primitivos (stack), os valores são passados por valor, o que significa que cópias dos valores são empilhadas na stack de chamadas. Qualquer modificação feita na função não afeta as variáveis originais fora dela, pois cada chamada da função tem sua própria cópia dos valores.

- Na função **sum2**, que opera com objetos (heap), as referências aos objetos são passadas por referência. Isso significa que apenas um ponteiro para o objeto é passado na stack de chamadas, não uma cópia do objeto em si. Como resultado, a função pode modificar diretamente o objeto referenciado, e qualquer alteração é refletida fora da função, já que ambas as referências (dentro e fora da função) apontam para o mesmo objeto no heap.

Essa diferença no comportamento de passagem de valor ou referência é influenciada pela alocação de memória no heap (objetos) e na stack (primitivos) em JavaScript, afetando a forma como os dados são manipulados e compartilhados entre funções.

# Módulo 10 – O objeto como um tipo (1 / 5)

No contexto da linguagem de programação JavaScript, é importante compreender os diferentes tipos de dados, com destaque para o comportamento especial dos objetos. Os tipos de dados desempenham um papel fundamental na programação, influenciando a forma como os valores são armazenados e manipulados.
JavaScript oferece tipos de dados primitivos, como o **tipo Number (número) e strings de texto (string)**, que representam informações de forma simples e direta:

```
const variavel1 = 2;      // Um número do tipo Number
const variavel2 = '2';    // Uma string de texto do tipo string
```

Esses tipos primitivos possuem operações específicas, como a soma de números ou a concatenação de strings.
Por outro lado, os **objetos** em JavaScript são tipos de dados complexos que podem armazenar informações em pares de **chave-valor**, como exemplificado a seguir:

```
const obj1 = {
    nome: 'Daniel',
    endereco: 'rua que sobe e desce',
};
```

Um aspecto notável dos objetos em JavaScript é a sua versatilidade, pois podem conter valores de qualquer tipo, inclusive outros objetos e funções.
Contudo, é importante ressaltar que, em JavaScript, objetos são passados por referência, o que significa que as manipulações realizadas em um objeto afetam diretamente a instância original. Isso difere dos tipos primitivos, que são passados por valor, criando cópias independentes.

# Módulo 10 – O objeto como um tipo (2 / 5)

Nesse momento, vamos criar um construtor chamado **BankAccount**. Ele será responsável por criar objetos que representarão contas bancárias em nosso programa. Para começar, vamos criar uma função que servirá como a base do nosso construtor. Essa função receberá três parâmetros: **personName** para o nome da pessoa que detém a conta, **initialBalance** para o saldo inicial da conta e **address** para o endereço associado a essa conta.

Dentro da função, usaremos a palavra-chave **this** para criar e definir as propriedades do objeto que estamos prestes a criar. Usaremos **this.personName = personName** para definir a propriedade **personName** do objeto como o valor passado como **personName** durante a criação. Faremos o mesmo para **initialBalance** e **address**.

Assim, quando utilizarmos esse construtor, ele nos permitirá criar facilmente objetos de conta bancária, preenchendo as informações necessárias, como nome da pessoa, saldo inicial e endereço. Isso torna a criação e manipulação de objetos de conta bancária mais eficiente e organizada em nosso código. Portanto, o construtor **BankAccount** é uma ferramenta valiosa para trabalhar com informações de contas bancárias em nossos programas em JavaScript.

```
function BankAccount(personName, initialBalance, address) {
    this.personName = personName;
    this.initialBalance = initialBalance;
    this.address = address;
}
```

# Módulo 10 – O objeto como um tipo (3 / 5)

Agora que criamos nosso construtor **BankAccount**, podemos usá-lo para criar instâncias individuais de contas bancárias. Primeiro, declaramos uma nova variável chamada **acc1** e a inicializamos com uma instância do objeto **BankAccount**. Usamos a palavra-chave **new** seguida pelo nome do construtor **BankAccount**. Em seguida, passamos três argumentos entre parênteses: **'Daniel'** para o nome da pessoa, **200** para o saldo inicial e **'rua sobe e desce'** para o endereço. Esses valores são exatamente os parâmetros que nossa função construtora espera.

Agora, **acc1** é um objeto de conta bancária com todas as propriedades definidas em nosso construtor, ou seja, **personName**, **initialBalance**, e **address**. Portanto, se nós imprimirmos **acc1** no console com **console.log(acc1)**, veremos essas propriedades e seus valores.

Além disso, para verificar o tipo de **acc1**, usamos **typeof acc1**. Como resultado, obtemos **'object'**, o que indica que **acc1** é um objeto JavaScript, como esperado.

```
const acc1 = new BankAccount('Daniel', 200, 'rua sobe e desce');
console.log(acc1);
console.log(typeof acc1);
```

```
/Users/danielporto/.nvm/versions/node/v15.13.0/bin/node ./objectDataTypes.js
> BankAccount {personName: 'Daniel', initialBalance: 200, address: 'rua que sobe e
sce'}
object
```

# Módulo 10 – O objeto como um tipo (4 / 5)

É importante notar que o objeto **acc1**, criado a partir do nosso construtor **BankAccount**, não é apenas do tipo genérico "objeto" em JavaScript. Na verdade, podemos interpretá-lo como um "objeto de conta bancária". Isso significa que ele possui propriedades específicas, como **personName**, **initialBalance** e **address**, que são características intrínsecas a uma conta bancária.

```
const acc1 = new BankAccount('Daniel', 200, 'rua que sobe e desce');
console.log(acc1);
console.log(typeof acc1);

const acc2 = new BankAccount('Lira', 200000, 'Broadway');
console.log(acc2);
```

```
/Users/danielporto/.nvm/versions/node/v15.13.0/bin/node ./objectDataTypes.js
> BankAccount {personName: 'Daniel', initialBalance: 200, address: 'rua que sobe e
sce'}
object
> BankAccount {personName: 'Lira', initialBalance: 200000, address: 'Broadway'}
```

Essa é uma das vantagens da programação orientada a objetos em JavaScript. Podemos criar tipos personalizados de objetos que representam entidades do mundo real, como contas bancárias neste caso. Isso torna nosso código mais legível e semântico, uma vez que podemos entender intuitivamente que **acc1** não é apenas um objeto genérico, mas sim uma instância de uma conta bancária com todas as informações necessárias.

Portanto, ao trabalhar com JavaScript, é fundamental não apenas reconhecer que temos objetos, mas também entender o tipo específico de objeto que estamos lidando. Essa abordagem torna nosso código mais claro e ajuda a manter a organização e a compreensão do que está acontecendo em nossos programas.

# Módulo 10 – O objeto como um tipo (5 / 5)

Em JavaScript, embora não tenhamos o conceito de classes no sentido tradicional, temos uma maneira de criar estruturas de objetos reutilizáveis chamadas construtores. Os construtores desempenham um papel semelhante ao de classes em outros paradigmas de programação orientada a objetos.

Em JavaScript, a instância de objetos ocorre quando você utiliza a palavra-chave **new** seguida do nome de um construtor, como **new BankAccount(...)**. Nesse contexto, **BankAccount** é o construtor que criamos para representar contas bancárias. Essa abordagem é semelhante ao conceito de classes em outras linguagens de programação orientada a objetos.

Quando você chama **new BankAccount('Daniel', 200, 'rua sobe e desce')**, o JavaScript cria uma nova instância de objeto com base no modelo definido pelo construtor **BankAccount**. Esta instância, que podemos chamar de **acc1**, herda todas as propriedades e métodos definidos no construtor, neste caso, **personName**, **initialBalance** e **address**.

Portanto, **acc1** é uma instância de uma conta bancária com os valores específicos passados durante a criação, ou seja, **'Daniel'** para **personName**, **200** para **initialBalance** e **'rua sobe e desce'** para **address**. Essa capacidade de criar várias instâncias de objetos com base em um construtor comum é uma maneira poderosa de organizar e reutilizar código, seguindo um paradigma de programação orientada a objetos em JavaScript.

E essas instâncias irã ser referenciadas muitas vezes como classes de um tipo de objeto.

```
This constructor function may be converted to a class declaration. ts(80002)
function BankAccount(personName: any, initialBalance: any, address: any): void
Quick Fix... (Ctrl+.)
function BankAccount(personName, initialBalance, address) {
    this.personName = personName;
    this.initialBalance = initialBalance;
    this.address = address;
}

const acc1 = new BankAccount('Daniel', 200, 'rua sobe e desce');
console.log(acc1);
console.log(typeof acc1);
```

# Módulo 10 – O operador instanceof (1 / 3)

Nessa aula vamos iniciar com um código parecido com o que construímos na aula anterior que cria uma classe chamada **BankAccount** (Conta Bancária) para representar informações sobre contas bancárias.

Vou explicar o código passo a passo relembrando alguns conceitos vistos anteriormente:

**Declaração da Classe BankAccount**:

- A classe **BankAccount** é definida usando a função construtora. Funções construtoras em JavaScript são usadas para criar objetos com propriedades e métodos associados.

**Parâmetros do Construtor**:

- O construtor da classe **BankAccount** aceita três parâmetros: **personName** (nome da pessoa), **initialBalance** (saldo inicial) e **address** (endereço).

```
function BankAccount(personName, initialBalance, address) {
    this.personName = personName;
    this.balance = initialBalance;
    this.address = address;

    this.printInfo = function () {
        console.log(
            `Essa conta pertence a ${this.personName}, residente do endereço '${this.address}'
            e possui um saldo atual de ${Number(this.balance).toFixed(2)} reais`
        );
    };
}

const ba1 = new BankAccount('Daniel', 200, 'Rua que sobe e desce');

ba1.printInfo();
```

**Atribuição de Propriedades**:

- Dentro do construtor, os valores passados como parâmetros são atribuídos às propriedades da instância da classe (**this.personName**, **this.balance** e **this.address**).

# Módulo 10 – O operador instanceof (2 / 3)

**Método printInfo**:

- A classe **BankAccount** possui um método chamado **printInfo**. Este método é uma função que imprime informações sobre a conta bancária no console.
- O método utiliza a interpolação de strings (template literals) para criar uma mensagem formatada que inclui o nome da pessoa, o endereço e o saldo atual.
- O saldo é formatado com duas casas decimais usando **Number(this.balance).toFixed(2)** para torná-lo mais legível.

**Criação de uma Instância da Classe BankAccount**:

- Após a definição da classe, uma instância da classe **BankAccount** é criada com o nome **ba1**. Esta instância representa uma conta bancária com o nome "Daniel", um saldo inicial de 200 reais e o endereço "Rua que sobe e desce".

**Chamada do Método printInfo**:

- Finalmente, o método **printInfo** é chamado na instância **ba1**, o que resulta na impressão das informações da conta bancária no console.

```
C:\Program Files\nodejs\node.exe .\instanceof.js
Essa conta pertence a Daniel, residente do endereço 'Rua que sobe e desce' e possui um saldo atual de 2000.00 reais.
```

# Módulo 10 – O operador instanceof (3 / 3)

```
console.log(typeof ba1);
console.log(ba1 instanceof BankAccount);
```

O operador **typeof** em JavaScript é usado para determinar o tipo de dado de uma variável ou expressão. No contexto de **typeof ba1**, ele retorna o tipo de dados da variável **ba1**, que será "object", pois **ba1** é uma instância de um objeto da classe **BankAccount**.

A expressão **ba1 instanceof BankAccount** é chamada de operador **instanceof** em JavaScript. Ela verifica se o objeto à esquerda (**ba1**) é uma instância do construtor à direita (**BankAccount**), retornando **true** se **ba1** foi criado usando **new BankAccount(...)**, indicando que herda propriedades e métodos da classe **BankAccount**, e **false** se não é uma instância dessa classe. Essa verificação é útil para determinar o tipo de objeto em tempo de execução.

```
C:\Program Files\nodejs\node.exe .\instanceof.js
Essa conta pertence a Daniel, residente do endereço 'Rua que sobe e desce' e possui um saldo atual de 2000.00 reais.
object  ← typeof
true    ← instanceof
```

# Módulo 10 – Classes em Javascript (1 / 3)

Em JavaScript, as **classes** são como moldes ou formas que permitem criar objetos com características e comportamentos específicos. Elas definem a estrutura e o comportamento que os objetos devem ter, fornecendo um conjunto de propriedades e métodos que os objetos criados a partir delas herdam.

Uma **classe** em JavaScript serve como um plano mestre que descreve como os objetos devem ser construídos. Ela define as propriedades (dados) que os objetos terão e os métodos (funções) que esses objetos poderão executar. Quando você instancia uma classe usando o operador **new**, você está criando um objeto que segue o modelo definido pela classe.

O JavaScript oferece suporte à criação e manipulação de classes como parte de sua programação orientada a objetos (POO). Foi introduzido uma sintaxe mais amigável para definir classes a partir da versão ES6 (ECMAScript 2015). Agora você pode usar a palavra-chave **class** para declarar uma classe e definir construtores, propriedades e métodos de forma mais declarativa.

Para definir uma **classe** em JavaScript, você utiliza a palavra-chave **class**, seguida pelo nome da classe em conformidade com a convenção PascalCase. Dentro das chaves **{}**, você pode declarar propriedades e métodos que desejam que a classe tenha. Isso cria um molde ou um modelo que pode ser usado posteriormente para criar objetos com características e comportamentos específicos.

```
class NomeDaClasse {
  // Propriedades e métodos da classe podem ser definidos aqui
}
```

# Módulo 10 – Classes em Javascript (2 / 3)

Agora, vamos transformar a função construtora **bankAccount** em uma classe. Isso é uma ação interessante de se realizar porque ao transformar a função construtora em uma classe, estaremos organizando nosso código de uma maneira mais moderna e estruturada. As classes fornecem uma abordagem mais clara e fácil de entender para criar objetos com propriedades e comportamentos específicos. Isso torna nosso código mais legível, reutilizável e alinhado com as melhores práticas de programação orientada a objetos em JavaScript.

Dentro de uma classe em JavaScript, o método **constructor** é um especial que é chamado automaticamente quando uma nova instância da classe é criada usando **new**. Ele é usado para inicializar as propriedades do objeto recém-criado, permitindo configurar seu estado inicial. Essa função é opcional, mas útil para definir o comportamento padrão dos objetos da classe.

```
class BankAccount {
    constructor() {

    }
}
```

```
class BankAccount {
    constructor(personName, initialBalance,address) {
        this.personName = personName;
        this.balance = initialBalance;
        this.address = address;
    }

    printInfo() {
        console.log(
            `Essa conta pertence a ${this.personName}, residente do endereço '${this.address}'
            e possui um saldo atual de ${Number(this.balance).toFixed(2)} reais`
        );
    }
}
```

Vamos analisar como estamos criando a nossa classe, suas propriedades e como adicionamos métodos a ela. Utilizando o código ao lado:

- **class BankAccount { … }**: Isso define a classe **BankAccount**.
- **constructor(personName, initialBalance, address) { … }**: Este é o construtor da classe. Ele é chamado quando você cria uma nova instância da classe **BankAccount**. O construtor aceita três argumentos: **personName** (nome da pessoa), **initialBalance** (saldo inicial) e **address** (endereço). Dentro do construtor, esses valores são atribuídos às propriedades da instância (**this.personName**, **this.balance** e **this.address**).

- **printInfo() { ... }**: Este é um método da classe **BankAccount**. Ele permite imprimir informações sobre a conta bancária no console. A mensagem de saída é formatada usando interpolação de strings (template literals) e exibe o nome da pessoa, o endereço e o saldo atual, formatado com duas casas decimais.

Dentro de uma classe em JavaScript, os métodos não precisam ser criados com a palavra-chave **function** porque a sintaxe da classe introduz uma maneira mais concisa e legível de definir métodos. Ao declarar um método dentro de uma classe, você simplesmente o define como uma função, sem a necessidade de usar a palavra-chave **function**. Isso torna o código mais limpo e mais próximo da linguagem natural.

```
function MinhaClasse() {
    this.meuMetodo = function() {
        // Implementação do método aqui
    };
}
```

**Utilizando função construtora**

```
class MinhaClasse {
    meuMetodo() {
        // Implementação do método aqui
    }
}
```

**Utilizando classe**

Ambas as abordagens fazem a mesma coisa, mas a segunda é mais legível e considerada uma prática recomendada em JavaScript moderno.

# Módulo 10 – Classes vs Construtor: As várias versões do JS (1 / 4)

Em JavaScript, a palavra-chave **class** é usada para definir uma classe, mas, quando você verifica o tipo de uma classe com o operador **typeof**, o resultado é **"function"**. Isso pode parecer um pouco confuso, mas há uma explicação para isso.
Essa aparente discrepância ocorre porque, em JavaScript, as classes são, na verdade, funções construtoras avançadas. Quando você define uma classe, ela cria uma função construtora por baixo dos panos. Esta função construtora é usada para criar instâncias da classe quando você usa o operador **new**.

Por exemplo, considere a seguinte classe:

```
class MinhaClasse {
  constructor() {
    // Construtor da classe
  }

  meuMetodo() {
    // Método da classe
  }
}
```

Por trás dessa classe, existe uma função construtora equivalente que poderia ser escrita assim:

```
function MinhaClasse() {
  // Construtor da classe
}

MinhaClasse.prototype.meuMetodo = function() {
  // Método da classe
};
```

Quando você verifica o tipo de **MinhaClasse** com **typeof MinhaClasse**, o resultado é **"function"** porque, internamente, **MinhaClasse** é uma função.
Portanto, a palavra-chave **class** é uma abstração de alto nível que simplifica a criação de construtores e métodos em JavaScript, mas, em última análise, ainda se baseia no uso de funções para criar objetos.

# Módulo 10 – Classes vs Construtor: As várias versões do JS (2 / 4)

Nesse momento vamos abrir um navegador, e pesquisar no Google por **javascript compatibility chart** e entrar no site **"Can I use"**

**"Can I Use" (https://caniuse.com/)** é um site popular usado por desenvolvedores web e designers para verificar a compatibilidade de recursos e tecnologias da web em diferentes navegadores da web. Ele fornece informações detalhadas sobre o suporte a recursos HTML, CSS, JavaScript e outras tecnologias em navegadores específicos, permitindo que os desenvolvedores determinem se podem usar determinados recursos em seus projetos ou se precisam fornecer alternativas ou ajustes para lidar com diferentes navegadores.

O **"Can I Use"** é uma ferramenta que oferece:

- Pesquisa de recursos para verificar a compatibilidade com navegadores.
- Tabelas coloridas que mostram o suporte a recursos em diferentes navegadores.
- Histórico de versões de navegadores para entender quando recursos foram introduzidos ou alterados.
- Notas de compatibilidade e informações adicionais.
- Filtros e opções de visualização personalizados.
- Integração com ferramentas de desenvolvimento web.

# Módulo 10 – Classes vs Construtor: As várias versões do JS (3 / 4)

Nesse momento vamos abrir um navegador, e pesquisar no Google por **babel transpiler** e entrar no site **"Babel.js"**

**"Babel.js"** (https://babeljs.io/)é um compilador de JavaScript que permite que desenvolvedores escrevam código moderno em JavaScript (como ES6/ES2015 ou versões mais recentes) e o compile para versões mais antigas de JavaScript que sejam compatíveis com a maioria dos navegadores.

O "**Babel.js"** é uma ferramenta essencial no desenvolvimento web devido a:

- Habilitar o uso de recursos JavaScript modernos, mesmo em ambientes que não os suportam nativamente.
- Oferecer alta personalização para adaptar a compilação do código às necessidades do projeto.
- Integrar-se facilmente em um ecossistema de ferramentas amplamente utilizado, como Webpack e ESLint.
- Permitir extensões de funcionalidade por meio de plugins e presets.
- Realizar transformações no código-fonte para garantir compatibilidade com navegadores mais antigos e ambientes específicos.
- Contar com uma comunidade ativa de desenvolvedores, garantindo suporte e recursos acessíveis.
- Facilitar a adoção rápida de novas tecnologias JavaScript, garantindo compatibilidade em diversos ambientes.

# Módulo 10 – Classes vs Construtor: As várias versões do JS (4 / 4)

O Babel é um transpilador essencial no mundo do desenvolvimento web. Ele desempenha um papel crucial ao permitir que os desenvolvedores escrevam código em versões mais recentes do JavaScript, como ES6 (também conhecido como ES2015) e além, e o converta em código compatível com versões mais antigas, como ES5.

Agora vamos clicar na aba "TRY OUT", que fornece um editor de código onde você pode inserir seu próprio código JavaScript ou código em versões modernas do JavaScript, como ES6 ou ESNext. O Babel então transpila automaticamente esse código para uma versão mais antiga, como ES5, e exibe o resultado em tempo real.

Em Babel.js, o **"target"** se refere ao ambiente ou versão do JavaScript para o qual você deseja compilar seu código-fonte. É uma configuração que permite direcionar a saída gerada pelo Babel para que seja compatível com navegadores ou ambientes específicos.

Vamos começar esta aula criando um pequeno projeto. Neste projeto, vamos construir uma classe chamada **"Cronômetro",** que é conhecida como **"stopwatch"** em inglês. Para iniciarmos precisamos entender como que o cronômetro funciona para construirmos as funcionalidades dele.

Um **cronômetro** funciona registrando o tempo decorrido entre o início e a parada, utilizando uma base de tempo interna (geralmente segundos) para calcular a diferença. O usuário inicia o cronômetro e, em seguida, para quando deseja medir o intervalo de tempo.

Antes de colocarmos a mão na massa e criar passo a passo do nosso "Cronômetro", mostrarei uma prévia de como essa nossa classe irá se comportar.

Para criarmos a nossa classe iremos utilizar a **sintaxe new Stopwatch()**, que será armazenada em uma variável, nesse caso irei o utilizar o seguinte código:

```
> const sw1 = new Stopwatch();
```

Lembrando de alguns conceitos, o uso da sintaxe **new Stopwatch(),** cria uma nova instância da classe "Stopwatch" usando o operador **new**. Essa linha de código chama o construtor da classe "Stopwatch", inicializando um objeto de cronômetro. O resultado dessa expressão é um objeto que possui as propriedades e métodos definidos na classe "Stopwatch".

# Módulo 10 – Programando em dupla: Apresentação do nosso projeto (2 / 2)

O objeto "sw1" possui propriedades e métodos globais, como também incluirá métodos adicionais que iremos criar, como "reset," "start," e "stop." Esses métodos globais e personalizados permitem controlar o cronômetro de acordo com as necessidades específicas do projeto.

```
> const sw1 = new Stopwatch();
<· undefined
> sw1.
<· __defineGetter__          tab
   __defineSetter__
   __lookupGetter__
   __lookupSetter__
   __proto__
   constructor
   elapsedTime
   hasOwnProperty
   isPrototypeOf
   propertyIsEnumerable
   reset
   start
   stop
   toLocaleString
   toString
   valueOf
>                sw1.elapsedTime
```

Os métodos servirão para iniciar, parar, zerar e obter o tempo decorrido do cronômetro.

```
> const sw1 = new Stopwatch();
<· undefined
> sw1.elapsedTime
<· '00:00:00'
> sw1.start()
<· undefined
> sw1.elapsedTime
<· '00:00:16'
> sw1.elapsedTime
<· '00:00:19'
>
```

# Módulo 10 – O construtor padrão do Cronômetro (1 / 3)

Agora é hora de colocar a mão na massa, e vamos iniciar criando um novo arquivo chamado **Stopwatch.js**. A convenção de nomear arquivos em maiúsculas, como "Stopwatch.js," é uma prática comum em muitos projetos JavaScript e em linguagens orientadas a objetos em geral. Isso é feito principalmente para indicar que o arquivo contém a definição de uma classe (ou construtor) que pode ser usada para criar objetos.

```
JS Stopwatch.js > Stopwatch
1   class Stopwatch {
2
3   }
4
```

Dentro da nossa classe, vamos iniciar um construtor e adicionar o esqueleto dos métodos que criaremos para que o nosso cronômetro tenhas todas as funcionalidades.

```
1   class Stopwatch {
2     constructor() {}
3
4     start() {}
5
6     stop() {}
7
8     reset() {}
9   }
```

# Módulo 10 – O construtor padrão do Cronômetro (2 / 3)

Mas observe que falta a propriedade que armazena o tempo do nosso cronômetro,  o que nos faz refletir é onde deveríamos inicializar ela. Para isso, precisamos relembrar o conceito de constructor.

O **constructor{}** é um método especial dentro de uma classe em JavaScript que é executado automaticamente quando um objeto é criado a partir dessa classe. Ele é usado para inicializar as propriedades do objeto e pode aceitar argumentos para configurar o objeto durante a criação, mas quando não precisamos da informação vinda por um usuário, não precisamos iniciar essa propriedade dentro do construtor, pois sabemos que o estado inicial do nosso cronômetro será 00:00:00. Portanto, podemos criar essa condição fora do construtor:

```
JS Stopwatch.js > Stopwatch
1   class Stopwatch {
2     elapsedTime = 0;
3
4     constructor() {}
5
6     start() {}
7
8     stop() {}
9
10    reset() {}
11  }
```

Definir propriedades fora do construtor, como **elapsedTime = 0;**, é uma prática comum em JavaScript quando você deseja fornecer um valor padrão para uma propriedade de objeto que pode ser inicializada posteriormente ou que não depende de argumentos fornecidos durante a criação do objeto.

No nosso código, estamos definindo a propriedade **elapsedTime** com um valor padrão de **0**. Isso significa que quando um objeto da classe é criado, essa propriedade é automaticamente inicializada com zero, a menos que seja explicitamente alterada posteriormente. Isso é útil quando você quer que a propriedade comece com um valor específico, mas ainda deseja a flexibilidade de modificá-la mais tarde no código, se necessário.

Isso pode ser particularmente útil para propriedades que não dependem de entradas externas ou cálculos complexos durante a inicialização, mas ainda são parte importante do objeto e precisam de um valor inicial conhecido.

# Módulo 10 – O construtor padrão do Cronômetro (3 / 3)

O construtor padrão, que é definido como **constructor() {}**, é uma parte essencial de uma classe em JavaScript, mesmo que esteja vazio. Ele desempenha um papel importante no processo de criação de objetos a partir dessa classe e na configuração inicial de propriedades.
Mesmo que o construtor esteja vazio, você pode decidir inicializar propriedades da classe com valores padrão no construtor ou definir métodos para realizar essa inicialização. Isso pode ser útil para garantir que todas as instâncias da classe tenham um estado inicial consistente e também para fornecer funcionalidades padrão. Ou seja, dentro da nossa classe, definimos que toda instância da classe Cronômetro, iniciará com a **propriedade elapsedTime** definida inicialmente e terá os **métodos start, reset e stop** como funcionalidades padrão.

```
JS Stopwatch.js > ...
1   class Stopwatch {
2     elapsedTime = 0;
3
4     start() {}
5
6     stop() {}
7
8     reset() {}
9   }
10
11  const sw1 = new Stopwatch();
12  console.log(sw1);
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   SQL CONSOLE
C:\Program Files\nodejs\node.exe .\Stopwatch.js
> Stopwatch {elapsedTime: 0}
```

```
JS Stopwatch.js > ...
1   class Stopwatch {
2     elap   Stopwatch {elapsedTime: 0}
3              elapsedTime: 0
4     star     [[Prototype]]: Object
5              constructor: class Stopwatch {\r\n  elapsedTime = 0;\r\n\r\n  start() {}
6     stop     reset: ƒ reset() {}
7              start: ƒ start() {}
8     rese     stop: ƒ stop() {}
9   }          [[Prototype]]: Object
10             Hold Alt key to switch to editor language hover
11  const sw1 = new Stopwatch();
12  console.log(sw1);
```

# Módulo 10 – Criando o Cronômetro (1 / 7)

Nomes específicos em variáveis e métodos tornam o código mais claro e legível. Facilitam a compreensão, manutenção e identificação de erros, promovendo boas práticas de programação. Então, vamos deixar mais claro ainda, que a nossa variável elapsedTime irá armazenar o valor de segundos dentro dela. E para isso vamos alterar a sua nomenclatura para **elapsedTimeInSeconds.**

```
elapsedTimeInSeconds = 0;
```

**Princípio da Abstração**

O **princípio da abstração** em uma classe JavaScript envolve a ideia de esconder detalhes internos e complexidades, expondo apenas as informações e funcionalidades essenciais para o uso externo. Isso é alcançado ao definir uma interface clara e métodos públicos que os usuários da classe podem acessar, enquanto os detalhes de implementação são mantidos internamente. A abstração permite a modularização e o gerenciamento mais eficiente de sistemas complexos, promovendo a simplicidade e reutilização de código.

# Módulo 10 – Criando o Cronômetro (2 / 7)

Vamos construir a funcionalidade do método start() e para isso vamos aprender uma nova função do Javascript, chamada setInterval().

A função **setInterval()** é uma função embutida do JavaScript que é usada para executar uma função ou um trecho de código repetidamente em intervalos regulares de tempo, ou seja, periodicamente a cada espaço de tempo definido na função. A sintaxe básica da função é:

```
setInterval(funcao, intervaloDeTempo);
```

Os parâmetros que a função recebe:

- **funcao**: A função que você deseja executar em intervalos regulares.
- **intervaloDeTempo**: O período de tempo em milissegundos entre cada execução da função.

```
class Stopwatch {
  elapsedTimeInSeconds = 0;

  start() {
    setInterval(() => {
      this.elapsedTimeInSeconds++;
    }, 1000);
  }
```

# Módulo 10 – Criando o Cronômetro (3 / 7)

Vamos criar e entender como funciona o nosso método start().

O método **start()** usa **setInterval()** para executar uma função a cada 1000 milissegundos (1 segundo).

A função que é executada a cada segundo incrementa a propriedade **elapsedTimeInSeconds** em 1, representando o aumento do tempo decorrido.

- Logo após iniciar o **Stopwatch**, o código exibe o valor de **sw1.elapsedTimeInSeconds**, que inicialmente é 0.
- Em seguida, usando **setTimeout()**, o código exibe novamente o valor de **sw1.elapsedTimeInSeconds** após 3 segundos. Neste ponto, o **setInterval()** já terá incrementado o **elapsedTimeInSeconds** algumas vezes, portanto, você verá um valor maior que 0, refletindo o tempo decorrido desde o início.

O resultado impresso no console mostrará o valor inicial como 0, seguido por um valor maior após 3 segundos, refletindo o tempo decorrido durante esse período.

```
class Stopwatch {
  elapsedTimeInSeconds = 0;

  start() {
    setInterval(() => {
      this.elapsedTimeInSeconds++;
    }, 1000);
  }
}

const sw1 = new Stopwatch();
sw1.start();
console.log(sw1.elapsedTimeInSeconds);
setTimeout(() => {
  console.log(sw1.elapsedTimeInSeconds);
}, 3000);
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   SQL CONSOLE
C:\Program Files\nodejs\node.exe .\Stopwatch.js
0
2
```

# Módulo 10 – Criando o Cronômetro (4 / 7)

A função **setInterval()** é frequentemente usada para executar uma determinada ação repetidamente em intervalos regulares de tempo. No entanto, há momentos em que você pode precisar interromper esse processo.

A função **setInterval()** retorna um valor chamado **"identificador de intervalo"** ou **"interval ID"** que pode ser armazenado em uma variável. Esse interval ID é usado para identificar o intervalo criado por **setInterval()**, permitindo que você o pare posteriormente usando uma função, que será o nosso **método stop()**. Para aplicarmos isso, vamos armazenar o retorno de setInterval() em uma variável chamada **intervalID.**

```
start() {
  intervalId = setInterval(() => {
    this.elapsedTimeInSeconds++;
  }, 1000);
}
```

A atribuição inicial de **null** ao **intervalId** é importante para garantir um comportamento consistente. Imagine-o como um indicador de execução. Inicialmente, ao definir **intervalId** como **null**, estamos indicando que nenhum intervalo está em execução. Isso previne erros se o método **stop()** for chamado antes de iniciar o intervalo, evitando comportamentos inesperados. Quando iniciamos o intervalo, **intervalId** recebe um valor que identifica o intervalo em execução.

```
class Stopwatch {
  elapsedTimeInSeconds = 0;
  intervalId = null;
```

# Módulo 10 – Criando o Cronômetro (5 / 7)

JS Stopwatch.js > Stopwatch > start

```javascript
class Stopwatch {
  elapsedTimeInSeconds = 0;
  intervalId = null;

  start() {
    this.intervalId = setInterval(() => {
      this.elapsedTimeInSeconds++;
    }, 1000);
  }

  stop() {
    clearInterval(this.intervalId);
  }

  reset() {}
}

const sw1 = new Stopwatch();
sw1.start();
```

Dentro do método **stop()**, utilizamos a função **clearInterval(intervalId)** para interromper o intervalo identificado por **intervalId**. O **intervalId** é um valor único que identifica um intervalo específico criado com **setInterval()**. Quando chamamos **clearInterval(intervalId)**, estamos dizendo ao JavaScript para parar de executar a função associada a esse intervalo em intervalos regulares.

O **clearInterval** é fundamental para evitar que a função definida em **setInterval()** continue a ser executada indefinidamente. Ele permite um controle preciso sobre a execução periódica do código, interrompendo o processo quando necessário. Em seguida, o **intervalId** pode ser reutilizado ou definido como **null** para indicar que não há mais intervalo ativo, mantendo o código organizado e evitando conflitos de execução.

# Módulo 10 – Criando o Cronômetro (6 / 7)

Para visualizarmos melhor os métodos sendo executados, vamos criar um arquivo chamado index.html. Dentro desse arquivo com o atalho **! + ENTER** , vamos criar uma estrutura base de HTML.

Dentro da tag <body>, vamos utilizar a tag <script> e importar o nosso arquivo **Stopwatch.js:**

```
index.html > html > body > script
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      <script src="/Stopwatch.js"></script>
11    </body>
12  </html>
```

Vamos iniciar o Live Serve, clicando com o botão direito do mouse sobre o arquivo index.html ou utilizando o atalho ALT+L ALT+O. E abrirmos o Dev Tools com F12, e clicar na aba Console.

# Módulo 10 – Criando o Cronômetro (7 / 7)

Ao aplicarmos a chamada do objeto cronômetro, que está armazenado na variável **sw1**, ao utilizar o método **stop()**, ocorre a interrupção da contagem e o armazenamento dos dados de segundos até o ponto em que o **sw1** foi parado.

```
JS Stopwatch.js > Stopwatch > reset
1   class Stopwatch {
2     elapsedTimeInSeconds = 0;
3     intervalId = null;
4
5     start() {
6       this.intervalId = setInterval(() => {
7         this.elapsedTimeInSeconds++;
8       }, 1000);
9     }
10
11    stop() {
12      clearInterval(this.intervalId);
13    }
14
15    reset() {
16      this.elapsedTimeInSeconds = 0;
17    }
18  }
19
20  const sw1 = new Stopwatch();
21  sw1.start();
```

Agora vamos finalizar construindo o método reset().

A função **reset()** é projetada para redefinir o cronômetro, ou seja, zerar o tempo decorrido. A propriedade **elapsedTimeInSeconds** é redefinida para 0, reiniciando a contagem a partir de zero segundos, mas isso ocorrerá apenas se o cronômetro estiver ativo. Caso contrário, a função **reset()** não terá efeito, garantindo que o tempo seja redefinido apenas quando necessário.

# Módulo 10 – Membros Estáticos (1 / 4)

Até o momento, nossa classe **Stopwatch** tem funcionado retornando os segundos, como 1, 2, 3, 4, o que não é o formato ideal para um cronômetro. No entanto, podemos considerar que estabelecemos a base do nosso projeto, com a funcionalidade principal do cronômetro em funcionamento.

Agora, estamos prontos para elevar a qualidade do nosso cronômetro. Para isso, vamos introduzir uma melhoria importante. Em vez de acessar diretamente a propriedade **elapsedTimeInSeconds** usando **this**, vamos adicionar um novo método chamado **formatTime**. Esse método receberá um parâmetro **timeInSeconds** e nos permitirá exibir o tempo decorrido no formato de horas, minutos, segundos e milissegundos.

Com o método **formatTime**, poderemos manipular e exibir o tempo decorrido da maneira desejada, o que é essencial para a apresentação visual adequada de um cronômetro.

Ele será um método estático. Vamos usar a palavra-chave **static** em antes do método, o que indicará que o método é associado à própria classe, em vez de ser associado a instâncias específicas da classe (ou seja, objetos). Como resultado, métodos estáticos não têm acesso direto às propriedades ou ao estado das instâncias da classe (representado pelo **this**).

Resumindo, o método **formatTime** é estático e não usa **this** para acessar propriedades de instância, como **elapsedTimeInSeconds**. Em vez disso, ele recebe **timeInSeconds** como um parâmetro explícito. Isso significa que você pode chamar o método **formatTime** diretamente na classe **Stopwatch**, fornecendo o valor de **timeInSeconds** como argumento

```
sw1.formatTime();
```

INCORRETO

```
Stopwatch.formatTime();
```

CORRETO

# Módulo 10 – Membros Estáticos (2 / 4)

O **método formatTime** será responsável por formatar o nosso cronômetro em horas, minutos e segundos. E a estrutura inicial do nosso método será:

```
static formatTime(timeInSeconds) {
  const hours = Math.floor(timeInSeconds / 3600);
  const minutes = Math.floor((timeInSeconds % 3600) / 60);
  const seconds = timeInSeconds - hours * 3600 - minutes * 60;

  return `${hours}:${minutes}:${seconds}`;
}
```

1. **hours** (horas):
   - Variável que armazena o número de horas no tempo.
   - A operação **timeInSeconds / 3600** divide o tempo em segundos por 3600 (60 segundos * 60 minutos) para obter o número total de horas.
   - **Math.floor** arredonda o resultado para o valor inteiro mais próximo, garantindo que não haja parte decimal.

2. **minutes** (minutos):
   - Variável que armazena o número de minutos no tempo restante após a obtenção das horas.
   - Primeiro, calculamos o tempo restante após subtrair o tempo das horas usando **(timeInSeconds % 3600)**, que retorna o resto da divisão de **timeInSeconds** por 3600.
   - Em seguida, dividimos esse valor por 60 para obter o número de minutos.
   - Mais uma vez, **Math.floor** é usado para arredondar o resultado.

3. **seconds** (segundos):
   - Variável que armazena o número de segundos restantes após a obtenção das horas e minutos.
   - Calculada subtraindo-se o tempo das horas e minutos do tempo total em segundos.

Agora, vamos adicionar uma nova função ao nosso cronômetro. Em vez de exibir o tempo no formato '0:0:0', queremos que ele mostre no formato '00:00:00'. Para fazer isso, vamos criar uma função chamada **'zeroPadding()**'. Essa função terá a responsabilidade de adicionar zeros à esquerda quando necessário para que o formato fique correto.

**zeroPadding()** será um método estático, porque ele desempenha sua função sem depender de informações ou estados específicos da instância do cronômetro. Em vez disso, sua finalidade é fornecer uma funcionalidade geral de formatação de números, o que faz sentido torná-lo acessível diretamente na classe, sem a necessidade de criar uma instância do cronômetro.

```
static zeroPadding(originalNumber, desireAmountDigits = 2) {
  // Passo 1: Convertemos o número original em uma string.
  let stringNumber = String(originalNumber);

  // Passo 2: Calculamos quantos zeros são necessários para atingir o número desejado de dígitos.
  const zerosRequired = desireAmountDigits - stringNumber.length;

  // Passo 3: Verificamos se já temos o número de dígitos desejado ou mais.
  if (zerosRequired <= 0) {
    // Se já tivermos, não precisamos adicionar zeros, então retornamos a string original.
    return stringNumber;
  }

  // Passo 4: Se precisarmos adicionar zeros, entramos em um loop.
  for (let counter = 0; counter < zerosRequired; counter++) {
    // Dentro do loop, adicionamos um zero à esquerda da string.
    stringNumber = `0${stringNumber}`;
  }

  // Passo 5: Após adicionar todos os zeros, retornamos a string com o formato desejado.
  return stringNumber;
}
```

# Módulo 10 – Membros Estáticos (4 / 4)

Vamos a uma explicação detalhada dos passos:

1. Convertemos o número original em uma string usando **String(originalNumber)**. Isso nos permite trabalhar com os dígitos individualmente.
2. Calculamos a quantidade de zeros necessários para atingir o número desejado de dígitos subtraindo o comprimento da string atual do número original do número desejado de dígitos.
3. Verificamos se já temos o número de dígitos desejado ou mais. Se já tivermos, não precisamos adicionar zeros, então retornamos a string original.
4. Se precisarmos adicionar zeros, entramos em um loop. O loop é executado o número de vezes necessário para adicionar os zeros faltantes.
5. Dentro do loop, adicionamos um zero à esquerda da string usando a notação de template string.
6. Após adicionar todos os zeros, retornamos a string com o formato desejado, agora com o número de dígitos correto.

Resumidamente, essa função **zeroPadding()** recebe um número e o formata, adicionando zeros à esquerda para garantir que ele tenha o número desejado de dígitos. Agora aplicaremos dentro do nosso método **formatTime()**, para que o nosso cronômetro represente o tempo com dois dígitos para horas, minutos e segundos (por exemplo, "01:05:09" em vez de "1:5:9").

```
static formatTime(timeInSeconds) {
  const hours = Math.floor(timeInSeconds / 3600);
  const minutes = Math.floor((timeInSeconds % 3600) / 60);
  const seconds = timeInSeconds - hours * 3600 - minutes * 60;

  return `${Stopwatch.zeroPadding(hours)}:${Stopwatch.zeroPadding(minutes)}:${Stopwatch.zeroPadding(seconds)}`;
}
```

# Módulo 10 – Getters e Setters (1 / 5)

Nesta aula, vamos usar o código que já criamos em aulas anteriores, o qual se trata de uma classe que representa uma conta bancária. Vamos aprender como aplicar os conceitos de **'getters' e 'setters'** usando um exemplo prático dentro do nosso programa bancário. Isso nos permitirá entender como acessar e modificar informações dentro da conta bancária de uma maneira controlada e segura.

O nosso programa possui a classe **BankAccount** representa uma conta bancária com propriedades para nome da pessoa, saldo inicial e endereço. O método **printInfo()** exibe informações da conta, incluindo o nome, endereço e saldo formatado. Em seguida, é criada uma instância **ba1** da classe com os valores "Daniel" como nome, 2000 como saldo inicial e "Rua que sobe e desce" como endereço.

JS BankAccount.js > ...

```javascript
class BankAccount {
  constructor(personName, initialBalance, address) {
    this.personName = personName;
    this.balance = initialBalance;
    this.address = address;
  }

  printInfo() {
    console.log(
      `Essa conta pertence a ${this.personName}, residente do endereço '${
        this.address
      }' e possui um saldo atual de ${Number(this.balance).toFixed(2)} reais.`
    );
  }
}

const ba1 = new BankAccount('Daniel', 2000, 'Rua que sobe e desce');
```

# Módulo 10 – Getters e Setters (2 / 5)

Os **getters e setters** são métodos especiais em JavaScript que permitem controlar o acesso e a modificação de propriedades de objetos. Eles são usados para garantir a encapsulação e a segurança dos dados em objetos. Eles podem ser considerados como guardiões de regras para as propriedades de um objeto. Eles agem como intermediários que impõem regras específicas quando você tenta acessar ou modificar essas propriedades.

Um **getter, ou "acessador"**, permite que você defina regras para a obtenção de valores de propriedades. Por exemplo, você pode garantir que um valor seja sempre positivo ou formatado de uma determinada maneira.
Por outro lado, um **setter, ou "modificador"**, atua como um guardião que verifica e aplica regras quando você tenta alterar o valor de uma propriedade. Isso garante que os valores atribuídos atendam a critérios específicos, como limites ou validações.

**Getter (Acessador)**:

- Um getter é um método que é usado para obter o valor de uma propriedade de um objeto.
- Ele é chamado quando você tenta acessar a propriedade.
- É útil quando você deseja realizar alguma lógica ou validação antes de retornar o valor da propriedade.
- Declaração: **get nomeDoGetter() { /* código para obter o valor */ }**

**Setter (Modificador)**:

- Um setter é um método que é usado para definir o valor de uma propriedade de um objeto.
- Ele é chamado quando você tenta atribuir um valor a uma propriedade.
- É útil quando você deseja realizar alguma lógica ou validação antes de definir o valor da propriedade.
- Declaração: **set nomeDoSetter(novoValor) { /* código para definir o valor */ }**

# Módulo 10 – Getters e Setters (3 / 5)

Para aplicarmos os métodos getters e setters, vamos definir **_balance** com um valor inicial de zero, estabelecendo um valor padrão para essa propriedade. Isso significa que, caso nenhum valor seja especificado durante a criação de uma instância da classe, o **_balance** começará com zero.

Utilizar o prefixo **_ (underscore)** antes de nomes de variáveis dentro de uma classe é uma convenção comum em JavaScript e em muitas outras linguagens de programação. Essa convenção indica que a variável é considerada como uma **variável "privada"** ou de **uso interno dentro da classe**.

Vamos construir o getter da nossa classe BankAccount, como o seguinte código:

```
get balance() {
  return this._balance;
}
```

Essa estrutura define um getter chamado "balance" e retorna o valor da propriedade privada **_balance**.

# Módulo 10 – Getters e Setters (4 / 5)

Agora o método setter da nossa classe BankAccount terá a responsabilidade atualizar o valor do nosso _balance e aplicar as regras de negócio:

```
set balance(value) {
  if (Math.abs(this._balance - value > 2000)) {
    throw new Error(
      "Não permitimos saques ou depósitos maiores que 2000 reais"
    );
  }
  this._balance = value;
}
```

- Dentro desse método, temos uma condição: **"if (Math.abs(this._balance - value) > 2000)"**. Essa condição verifica se a diferença absoluta entre o **valor atual de "_balance"** e o novo valor **"value"** é maior que 2000 reais.

- Se essa condição for verdadeira, o código lança um erro com a mensagem "Não permitimos saques ou depósitos maiores que 2000 reais". Isso significa que estamos aplicando uma regra que impede saques ou depósitos maiores que 2000 reais de serem executados.
- Se a condição não for atendida, ou seja, se a diferença for menor ou igual a 2000 reais, o valor de "_balance" é atualizado com o novo valor "value".

Em resumo, esse método "set balance(value)" impõe uma regra de limite de 2000 reais para saques ou depósitos na propriedade "balance". Se a regra for violada, um erro é lançado; caso contrário, o valor é atualizado.

# Módulo 10 – Getters e Setters (5 / 5)

Agora, no nosso projeto do cronômetro, vamos usar o que aprendemos sobre **getters e setters**. Vamos adicionar o método get à nossa **classe Stopwatch**. Isso nos permitirá controlar e acessar melhor as informações do cronômetro, garantindo que funcionem corretamente de acordo com as nossas necessidades.

```
get elapsedTime() {
  return Stopwatch.formatTime(this.elapsedTimeInSeconds);
}
```

Então quando criamos a estrutura do nosso **método get**, criamos a funcionalidade para que ele de que quando você acessar a **propriedade "elapsedTime"** em uma instância da classe "Stopwatch", o código dentro deste getter será executado.

1. Dentro do getter, ele chama uma função chamada **"Stopwatch.formatTime(this.elapsedTimeInSeconds)"**.

2. Essa função **"Stopwatch.formatTime()"** é responsável por formatar o tempo decorrido em segundos (que é armazenado na propriedade **"elapsedTimeInSeconds"**) em um formato legível, como "00:00:00" (horas:minutos:segundos).

3. O valor formatado é retornado pelo getter como resultado. Portanto, quando você acessa "elapsedTime" em uma instância do cronômetro, ele retorna o tempo decorrido no formato desejado.

# Módulo 10 – Membros Privados (1 / 1)

Não criamos um **'setter'** para a nossa classe de Cronômetro porque não queremos que o usuário tenha a capacidade direta de alterar o valor de tempo do nosso Cronômetro. No entanto, é importante notar que, mesmo sem um 'setter', se o usuário chamar diretamente '**.elapsedTimeInSeconds**' e alterar seu valor, ele estará efetivamente realizando a mesma ação que um setter permitiria. No entanto, essa não é a nossa intenção, pois desejamos controlar e restringir como o tempo é manipulado na nossa classe.

E para isso, vamos aprender um novo conceito quando falamos em classes que são os membros privados.

**Membros privados** em classes são elementos de uma classe que só podem ser acessados internamente por métodos e atributos da própria classe. Eles são chamados de privados porque não podem ser acessados diretamente por objetos que instanciam a classe ou por classes derivadas (subclasses). Isso é uma forma de encapsulamento, que é um dos princípios da programação orientada a objetos.

A partir do ECMAScript 2019 (também conhecido como ES10 ou ES2019), JavaScript introduziu um novo recurso chamado **"private class fields"** (campos privados de classe) que utiliza o caractere **#** para criar membros verdadeiramente privados em classes.

O uso do **#** permite que os desenvolvedores criem membros privados de forma mais segura e intuitiva, sem depender de convenções de nomenclatura ou truques de escopo de classe. Isso melhora a encapsulação e ajuda a proteger os dados da classe contra acesso não autorizado.

```
#elapsedTimeInSeconds = 0;
#intervalId = null;
```

Nesse momento, nós vamos ir além e melhorar ainda mais o nosso projeto criando uma interface visual para o nosso Cronômetro.

Adicionaremos um parâmetro de **"callback"** que recebe uma **"arrow function vazia"** nos três métodos principais da nossa classe **start(),stop() e reset()**, lembrando que um callback é uma função que é passada como argumento para outra função e é executada posteriormente, geralmente em resposta a um evento ou ação específica.  Esse parâmetro será opcional e por isso que ele armazena uma arrow function vazia como default (valor padrão).

Havíamos criado a estrutura base de um arquivo html, chamado index.html e vamos trabalhar dentro dele para começar a criar a interface visual para o nosso naveagador.

```
start(callback = () => {}) {
  this.#intervalId = setInterval(() => {
    this.#elapsedTimeInSeconds++;
    callback();
  }, 1000);
}

stop(callback = () => {}) {
  clearInterval(this.#intervalId);
  callback();
}

reset(callback = () => {}) {
  this.#elapsedTimeInSeconds = 0;
  callback();
}
```

# Módulo 10 – Indo além: A representação visual do Projeto (2 / 5)

Dentro do nosso arquivo html, vamos adicionar alguns elementos  e seus respectivos IDs , além de algumas informações para visualizarmos no nosso navegador através da extensão Live Server, a base do nosso Cronômetro.

```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>Document</title>
  </head>
  <body>
    <section id="stopwatch-wrapper">
      <p>00:00:00</p>
      <div id="button-box">
        <button id="start">Iniciar</button>
        <button id="stop" class="secondary-button">Parar</button>
        <button id="reset">Reiniciar</button>
      </div>
    </section>
    <script src="/Stopwatch.js"></script>
  </body>
</html>
```

# Módulo 10 – Indo além: A representação visual do Projeto (3 / 5)

**1-** Agora vamos estilizar o nosso Cronômetro, criando um arquivo chamado style.css e importando dentro do arquivo index.html.

```
<head>
  <meta charset="UTF-8" />
  <meta http-equiv="X-UA-Compatible" content="IE
  <meta name="viewport" content="width=device-wi
  <link rel="stylesheet" href="./style.css" />
  <title>Daniel's Stopwatch</title>
</head>
```

**2 -** Vamos centralizar o nosso Cronômetro na página do navegador.

```
# style.css > body
1   body {
2     display: flex;
3     justify-content: center;
4     align-items: center;
5     height: 100dvh;
6   }
```

**3-** Vamos acessar o site chamado "Coolors" (https://coolors.co/). O Coolors é uma plataforma online popular para a criação de paletas de cores. Criaremos uma paleta de cor para o nosso projeto.

# Módulo 10 – Indo além: A representação visual do Projeto (4 / 5)

**4-** Agora vamos adicionar paleta de cores que escolhemos ao nosso projeto e já aplicaremos uma cor de fundo á nossa página.

```
# style.css > ...
:root {
  --stopwatch-primary-color: #033f63;
  --stopwatch-primary-shade: #05639e;
  --stopwatch-secondary-color: #e5a78a;
  --stopwatch-secondary-shade: #f0cdbc;
  --stopwatch-light-color: #e9f1f7;
  --body-background-color: #9cd7fc;
}

body {
  display: flex;
  justify-content: center;
  align-items: center;
  height: 100dvh;
  background-color: var(--body-background-color);
}
```

**5-** Vamos aplicar estilos padrão a todos os elementos da nossa página para definir a aparência geral. Se, por algum motivo, precisarmos mudar algum detalhe específico de estilo, faremos isso diretamente no elemento correspondente.

```
# style.css > *
* {
  box-sizing: border-box;
  margin: 0;
  padding: 0;
}
```

**6-** Vamos acessar o site oficial do Google Fonts : https://fonts.google.com/
Neste site, você pode explorar a vasta coleção de fontes tipográficas disponibilizadas pelo Google Fonts, visualizar amostras de texto com essas fontes e obter o código de incorporação para usá-las em seus projetos web.

# Módulo 10 – Indo além: A representação visual do Projeto (5 / 5)

**7-** Dentro do site Google Fonts, você pode pesquisar diversas fontes diferentes e ao lado direito o site nos mostra como importarmos para o nosso código, o formato no CSS e no HTML.

**8-** Implementaremos o código fonte que o site forneceu aos escolher as suas fontes e colocar dentro da tag <head> no arquivo index.html.

```
<link rel="preconnect" href="https://fonts.googleapis.com" />
<link rel="preconnect" href="https://fonts.gstatic.com" crossorigin />
<link
  href="https://fonts.googleapis.com/css2?family=Major+Mono+Display&f
  rel="stylesheet"
/>
```

```
* {
  box-sizing: border-box;
  margin: 0;
  padding: 0;
  font-family: 'Roboto', 'Raleway', sans-serif;
}
```

# Módulo 10 – Indo além: A representação visual do Projeto (5 / 5)

**9-** Agora vamos adicionar um ID para o nosso elemento <p> dentro do arquivo index.html.

```html
<section id="stopwatch-wrapper">
  <p id="stopwatch-display">00:00:00</p>
  <div id="button-box">
    <button id="start">Iniciar</button>
    <button id="stop" class="secondary-button">Parar</button>
    <button id="reset">Reiniciar</button>
  </div>
</section>
```

10- Vamos adicionar um estilo de fonte para o elemento <p>.

```css
#stopwatch-display {
  font-family: 'Major Mono Display', 'Orbitron';
  background-color: #000000;
  color: #ffffff;
}
```

E nesse momento, teremos a interface do nosso Cronômetro da seguinte maneira:

# Módulo 10 – Indo além: Ajustes finais (1 / 6)

Vamos começar os ajustes finais da nossa página web na qual a interface que estamos criando é a da nossa classe Cronômetro.

**1 -** Vamos aumentar o tamanho dos nossos elementos:

```
body {
  display: flex;
  justify-content: center;
  align-items: center;
  height: 100dvh;
  background-color: var(--body-background-color);
  font-size: 3rem;
}
```

**2 -** Vamos resetar todas as características de estilos dos botões:

```
button {
  all: unset;
}
```

**3 –** Adicionaremos alguns estilos aos botões, como bordas, espaçamento, cursor:

```
button {
  all: unset;
  cursor: pointer;
  border: 2px solid var(--stopwatch-light-color);
  padding: 0.5rem;
  border-radius: 10px;
}
```

**4 –** Agora criando os estilos para o nosso elemento de ID stopwatch-wrapper:

```
#stopwatch-wrapper {
  background-color: var(--stopwatch-primary-color);
  padding: 1rem;
  border-radius: 10px;
}
```

# Módulo 10 – Indo além: Ajustes finais (2 / 6)

Vamos começar os ajustes finais da nossa página web na qual a interface que estamos criando é a da nossa classe Cronômetro.

**5 -** Vamos alterar a cor de fundo dos nossos botões:

```
button {
  all: unset;
  cursor: pointer;
  border: 2px solid var(--stopwatch-light-color);
  padding: 0.5rem;
  border-radius: 10px;
  background-color: var(--stopwatch-secondary-color);
}
```

**6 –** No botão que representa o "Parar", vamos adicionar uma estilização específica, além de adicionar um estilo para quando o mouse passar por cima dos botões:

```
button.secondary-button {
  background-color: var(--stopwatch-primary-color);
  color: var(--stopwatch-secondary-color);
}

button:hover {
  background-color: var(--stopwatch-secondary-shade);
}

button.secondary-button:hover {
  background-color: var(--stopwatch-primary-shade);
}

button:focus {
  border: 5px solid #000000;
}
```

**4 –** Ajustaremos o nosso elemento de tempo para ficar centralizado e a caixa que envolve os nosso botões com margens específicas:

```
.button-box {
  margin: 1rem 0 0.5rem 0;
}

#stopwatch-display {
  font-family: 'Major Mono Display', 'Orbitron';
  background-color: #000000;
  color: #ffffff;
  text-align: center;
  padding: 1rem 0;
  border-radius: 10px;
}
```

# Módulo 10 – Indo além: Ajustes finais (3 / 6)

Para concluir nosso arquivo CSS, vamos adicionar estilos especiais para telas menores. Isso é importante porque queremos que nosso cronômetro mantenha uma aparência adequada e não fique distorcido, independentemente do dispositivo usado, seja um celular, tablet ou notebook. Dessa forma, a interface do cronômetro sempre será exibida de maneira apropriada, não importa o tamanho da tela que o usuário esteja usando.

E para isso vamos utilizar um conceito de @media. **@media** é uma regra utilizada em CSS para aplicar estilos diferentes a elementos HTML com base nas características da mídia em que estão sendo exibidos, como o tamanho da tela, a orientação do dispositivo, a resolução, entre outros. Isso permite que você crie layouts responsivos e adapte o design de uma página da web para diferentes tipos de dispositivos e tamanhos de tela.
A regra **@media** é usada para especificar uma condição que deve ser atendida para que o conjunto de estilos dentro dela seja aplicado. A sintaxe básica é a seguinte:

```
@media tipo-de-midia e (condição) {
  /* Estilos a serem aplicados quando a condição for verdadeira */
}
```

- **tipo-de-midia** pode ser especificado como **screen** (tela), **print** (impressão), **all** (todos) e outros valores para indicar o tipo de mídia alvo.
- **condição** é uma expressão que define as características da mídia que devem ser verdadeiras para que os estilos dentro da regra **@media** sejam aplicados. Por exemplo, você pode verificar a largura da tela, a orientação do dispositivo ou a resolução.

# Módulo 10 – Indo além: Ajustes finais (4 / 6)

Então dentro do arquivo style.css adicionaremos a seguinte estrutura de código:

```
@media (max-width: 600px) {
  :root {
    font-size: 10px;
  }
}
```

A regra **@media (max-width: 600px)** em CSS é uma instrução condicional que especifica que os estilos dentro das chaves **{ ... }** devem ser aplicados apenas quando a largura da tela do dispositivo for igual ou menor que 600 pixels.

Neste caso específico, dentro dessa regra **@media**, você está alterando o valor da propriedade **font-size** da pseudo-classe **:root**.

A pseudo-classe **:root** representa o elemento raiz do documento HTML, geralmente associado ao elemento **<html>**. A propriedade **font-size** está sendo definida como **10px** quando a largura da tela for igual ou menor que 600 pixels.

# Módulo 10 – Indo além: Ajustes finais (5 / 6)

Voltando para o arquivo Stopwatch.js, vamos conectar os botões do Cronômetro com as funcionalidades da nossa classe.

```
52  const startBtn = document.getElementById('start');
53  const stopBtn = document.getElementById('stop');
54  const resetBtn = document.getElementById('reset');
55  const stopwatchDisplay = document.getElementById('stopwatch-display');
56
57  const sw1 = new Stopwatch();
```

Cada variável armazena um elemento de botão do HTML, que correspondem ao "Iniciar", "Parar", "Reiniciar". E também vamos armazenar o valor do nosso Display, ou seja, o elemento que mostra o tempo do Cronômetro.

Agora vamos criar os eventos que cada botão irá realizar, ou seja, os botões ao serem "clicados" terão uma funcionalidade disparada.

```
60  startBtn.addEventListener('click', () => {
61    sw1.start();
62  });
63  stopBtn.addEventListener('click', () => {
64    sw1.stop();
65  });
66  resetBtn.addEventListener('click', () => {
67    sw1.reset();
68  });
```

# Módulo 10 – Indo além: Ajustes finais (6 / 6)

Criaremos agora uma função para que as funcionalidades do nosso botão sejam projetadas dentro do display que mostra o tempo do nosso Cronômetro.  Essa função servirá como argumento para os nossos métodos, já que eles aceitam callbacks como parâmetros.

```
57  function updateDisplay() {
58    stopwatchDisplay.innerText = sw1.elapsedTime;
59  }
60
61  const sw1 = new Stopwatch();
62
63  startBtn.addEventListener('click', () => {
64    sw1.start(updateDisplay);
65  });
66  stopBtn.addEventListener('click', () => {
67    sw1.stop();
68  });
69  resetBtn.addEventListener('click', () => {
70    sw1.reset(updateDisplay);
71  });
72
```

Com tudo implementado nosso Cronômetro está funcionando corretamente!