

# RESUMÃÃO

Decola Tech: 3ª Edição

Alexandre Filho

Likedin: <https://www.linkedin.com/in/alexandre-psouza/>

Github: pessouza

(Isso é um material não-oficial. Qualquer sugestão ou feedback - caso apontem alguma incongruência - é bem-vindo.)

Bem-vindos(as)! Por meio desse resumo, pretendo juntar palavras e conceitos-chave abordados durante as aulas do Decola Tech 3. O conteúdo será atualizado à medida em que vou as assistindo. Simbora estudar!

## INTRODUÇÃO À PROGRAMAÇÃO E PENSAMENTO COMPUTACIONAL

### Fundamentos de Algoritmos

#### Tipos de dados:

- **Numérico:**  
Inteiros (Integer) (-2,-1,1,2,3,4...) e Reais (Float) (1.4,1.53,1.25...)
- **Caractere:**  
? A # K p e ! @  
Um conjunto de caracteres forma uma String.
- **Lógico (Booleano):**  
True, False

#### Variáveis:

Pode assumir qualquer um dos valores de um determinado conjunto de valores.

Regras:

- Atribuição de um ou mais caracteres
- Primeira letra - não número
- Sem espaços em branco
- Utilização de palavras reservadas
- Caracteres e números

Uma variável que é inalterável é uma **constante**, como, por exemplo,  $\pi=3,14$

#### Instruções primitivas:

As instruções executam um determinado tipo de ação para manipular um dado.

Cálculos matemáticos

Operador	Operação	Tipo	Prioridade Mat.	Tipo de Retorno de Resultado
+	Manutenção de sinal	Unário	1	Positivo
-	Inversão de Sinal	Unário	1	Negativo
↑	Exponenciação	Binário	2	Inteiro ou Real
/	Divisão	Binário	3	Real
div	Divisão	Binário	4	Inteiro
*	Multiplificação	Binário	3	Inteiro ou Real
+	Adição	Binário	4	Inteiro ou Real
-	Subtração	Binário	4	Inteiro ou Real

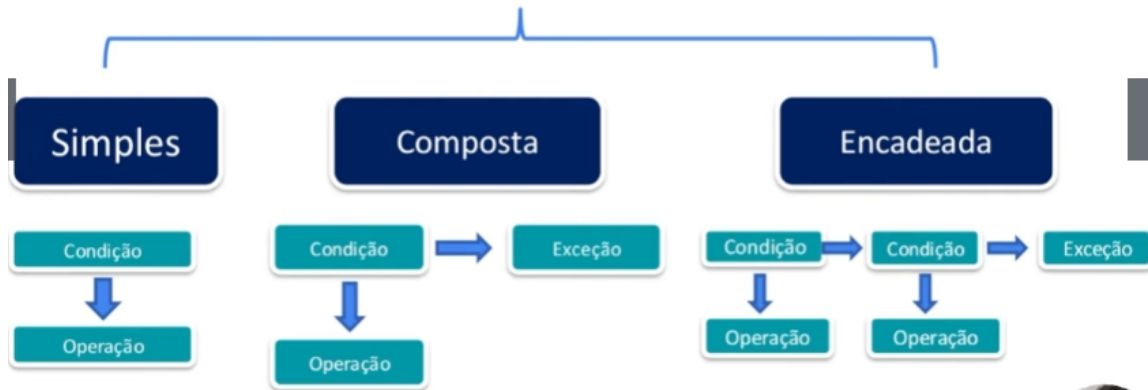
#### Estrutura condicional:

Algo deve atender a algum parâmetro determinado para que a condição seja considerada ou desconsiderada

- ex.: SE media<6 print="você foi reprovado"

Divide-se em **SIMPLES**, **COMPOSTA** ou **ENCADEADA**

## Estrutura Condicional



Alguns dos símbolos utilizados na estrutura condicional:

- = : Igual
- <> : Diferente
- > : Maior que
- < : Menor que
- >= : Maior ou igual a
- <= : Menor ou igual a

### Operadores lógicos:

Utilizamos estes operadores em questão de Verdadeiro e Falso.

- **And**  
Só será verdadeiro quando todas as condições forem verdadeiras  
ex: se (*gramática=aprovado* **and** *conversação=aprovado*), então escreva="aprovado". senão escreva="reprovado".
- **Or**  
Só precisa que uma condição seja verdadeira para ele ser verdadeiro  
ex: se *1= verdadeiro* **or** *2=falso*, escreva="verdadeiro"  
se *1=falso* **or** *2=falso*, escreva="falso"
- **Not**  
O not serve como um reversor de resultados  
ex: se algo é verdadeiro, o resultado é falso

### Estruturas de repetição:

Repetirá uma determinada ação até chegar num limite estipulado pelo código ou até uma condição ser satisfeita.

- Reduz o número de linhas de código
- Facilita compreensão
- Redução de erros

Existem mais de um tipo de estrutura de repetição:

- Enquanto .... faça ....
- Repita .... até ....
- Para .... de ..... até ..... faça ....

### Vetores e Matrizes:

**Vetor:** é caracterizado por uma variável dimensionada com tamanho pré-fixado unidimensional que irá receber *n* valores.

**Matriz:** é uma tabela organizada em linhas e colunas no formato m x n, onde m representa o número de linhas (horizontal) e n o número de colunas (vertical). Desta forma, matrizes são coleções de variáveis, contíguas em memória, com índices. ex.:

- nota 11=10  
nota 12=6

nota 21=10  
nota 22=4

notas\_aluno\_1=[10,6]  
notas\_aluno\_2=[10,4]

notas\_alunos=[10, 6, 10, 4...] a cada 2 posições, um novo aluno

Uma tabela de médias neste estilo é uma matriz, por ex.

Aluno	Nota 1	Nota 2	Nota 3	Média
1	10	6	8	8
1	8.5	7	9	8.17

### Funções:

As funções são blocos de instruções que **realizam tarefas específicas** ex:

- funcao mediaescolar(nota 1, nota2)

resultado=0  
resultado=(nota 1+ nota 2)/2

retorne resultado  
fim funcao

### Instruções de entrada/saída:

**Entrada:** Consiste na inserção e recebimento de dados do mundo real por meio de ação de alguma interface, seja teclado, mouse, arquivos, entre outros.

**Saída:** Consiste na impressão dos dados do mundo abstrato digital por meio de ação de alguma interface.

Existem dois tipos de saídas dentro de um programa (algoritmo):

- **Saída programada:**  
A saída programada pode ser condicional ou incondicional.  
A condicional aguarda o dispositivo para acionar a saída e imprimir os valores
- **Saída por interrupção:**  
Definida pelos periféricos - algo aconteceu no ambiente computacional que gerou uma interrupção, ocasionando a saída

Para toda saída haverá um caso, ela pode se apresentar como:

- Bem-sucedida
- Erro de sintaxe
- Erro de programação
- Problema na interface

## Linguagens de Programação

### Introdução às linguagens de programação:

A linguagem de programação é um método padronizado composto por um conjunto de regras sintáticas e semânticas de implementação de um **código fonte**.

Problemas computacionais são objetos de discussão que possuem instruções de passo a passo que são mais facilmente resolvíveis em ambiente computacional, sendo eles:

- **Problemas de decisão**  
Problemas de caráter lógico (sim ou não), com uma ideia de pertencimento
- **Problemas de busca**  
Possuem um relacionamento binário, com função de buscar algum elemento (ex.: x está em A?)
- **Problemas de otimização**  
Têm como objetivo maximizar ou minimizar alguma função, visando aperfeiçoamento.

### Como um computador entende o programa?:

#### Tradução:

Programa Fonte	Compilador	Programa Objeto	Linguagem de Máquina
----------------	------------	-----------------	----------------------

O **compilador traduz** o programa fonte para um programa objeto, que será executado pelo computador.

#### Características:

- Execução mais rápida
- Programas menores

ex: Java, C++...

#### Interpretação:

Programa Fonte	Linguagem de Máquina
----------------	----------------------

Na interpretação, o programa fonte é executado diretamente, tornando o processo mais lento.

#### Características:

- Maior flexibilidade

ex: JavaScript, Ruby, Python...

### Características de um programa:

Pode-se listar as seguintes principais características a serem consideradas:

- **Legibilidade:** ter uma boa leitura do código
- **Redigibilidade:** ter um código expressivo e de fácil escrita
- **Confiabilidade:** fazer o que o programa se propõe a fazer
- **Custo:** O código economiza recursos?

Além destas características, vale a pena considerar, também:

- Atualizações
- Onde uma linguagem é utilizada
- Disponibilidade de ferramentas
- Comunidade
- Adoção pelo mercado

### Análises de código:

O programa fonte, ao passar pelo compilador, será submetido a três processos:

- **Análise léxica:**

Divide-se nas seguintes ações:

- **Particionar:**  
Identifica os elementos e os agrupa: identificadores, palavras reservadas, números, strings, etc.
- **Classificar:**  
Analisa carácter por carácter, agrupando-os em símbolos
- **Eliminar:**  
Elimina caracteres em branco e comentários
- **Análise sintática:**  
Atribui às sentenças uma estrutura, definindo a corretude do programa.  
Dependerá da linguagem de programação utilizada
- **Análise semântica:**  
É o estudo do significado. O código faz o que é esperado?

### Paradigmas da programação:

Há uma série de paradigmas de programação:

- **Orientação a objeto:** baseia-se na utilização de objetos e suas interações  
Divide-se em:

O que tenho?	O que posso fazer?	Como faço?
Atributos	Métodos	Estados

A sua maior vantagem é o reuso de código

- **Procedural:** relacionado às ideias de sequências - chamadas sucessivas
- **Funcional:** relacionado às funções
- **Estruturado:** relacionado a blocos aninhados para problemas específicos e diretos
- **Computação distribuída:** relacionado a módulos independentes
- **Lógico:** muito utilizado em aplicações de inteligência artificial. Esse paradigma chega no resultado esperado a partir de avaliações lógico-matemáticas.

## PRIMEIROS PASSOS PARA DESENVOLVIMENTO WEB

O que é internet

### Termos-chave:

Download/Upload	Cache	Emoticon/Emoji	Spam	Backup
Navegador	Crack	Gif	HTML	Blog
Banda Larga	Email	Jpg/Png	HTTP	Cyberbullying
Link	MP3/MP4	Host	Ícone	3G/4G/5G
Login/Logon	Multimídia	Online/Offline	URL	Cookies
Logout/Logoff	Nick	Pixel	Vírus	Firewall
Keylogger	Phishing	Clickbait	Fake News	Podcast
Vlog	Pop-up	Youtuber	Hacker	IP

### A internet atualmente:

Em 2019:

**75%** dos brasileiros têm acesso à internet.

**90%** das pessoas acessam a internet todo dia

**92%** das pessoas utilizam Whatsapp, Skype, Messenger

**76%** utilizam Facebook/Snapchat

**73%** fazem chamada de vídeo por Whatsapp ou Skype

**68%** utilizam serviços eletrônicos

**58%** utilizam para enviar e-mails

**39%** das pessoas fazem compras pela internet

**11%** participam de listas e fóruns

### Como funciona a internet

#### O que são redes:

Interface de conexão entre computadores.

Os backbones são estruturas “parrudas”, espalhadas pelo mundo, que gerenciam o tráfego da internet e conexões.

Os provedores de acesso (geralmente empresas telefônicas) contratam sinal do backbone e repassam para o usuário final.

Estes provedores de acesso provêm serviços:

- Dial-up
- ADSL (banda larga)
- Fibra óptica
- Rádio
- Satélite
- Móvel
- P2P

Como funciona o acesso a um site?

1. Entra-se com o endereço (www....)
2. O servidor DNS (Domain Net Service) transforma o endereço num número, o IP  
Um exemplo de IP é o 127.0.0.1, que é o seu Local Host

### TCP/IP, portas, roteadores, switches e modems

#### TCP/IP e UDP:

**TCP:** Significa “Transmission Control Protocol - Protocolo de Controle de Transmissão”, são protocolos de comunicação entre computadores em rede.

**IP:** Significa “Protocolo de Internet”, e utiliza um modelo de camadas.

São quatro camadas:

- Física (ex.:placa de rede)
- Rede (ex.: IP)
- Transporte (ex.: TCP, UDP)
- Aplicação (ex.:FTP, SMTP, HTTP)

Diferença entre UDP e TCP:

**UDP:** Significa “User Datagram Protocol”, são protocolos de comunicação entre computadores em rede.

- Rápido
- Não confiável (não há confirmação de envio ou recebimento)
- “Carro do ovo” Não se sabe quem recebeu a mensagem
- *Livestream*

**TCP:**

- Voltado à conexão
- Handshake
- Integridade, ordem dos dados
- Aplicativo de mensagem de texto

### Portas:

São entradas onde os dados chegarão/sairão, ex:

- Porta 20: Utilizada para requisição FTP (para envio de arquivos)
- 22: SSH - Conexão segura entre computadores para executar comandos
- 25: SMTP - Envio de e-mails
- 53: DNS - Tradução de nome para IP
- 80: HTTP - Requisição simples de internet
- 443: HTTPS - Requisição segura de internet

### Roteadores, switches e modems:

#### Modem:

É abreviação de **Modulator-Demodulator** (modula e demodula sinais). É um hardware que converte dados em um formato que possa ser transmitido de um computador para outro e lido por outro.

#### Roteador:

Distribui internet para um ou mais dispositivos de uma rede, fazendo a comunicação entre redes. Envia sinais para todos os próximos, até para quem não os requisitou.

#### Switch:

Diferente do roteador, o switch vai distribuir internet àqueles que solicitaram.

## Celular, internet e outros dispositivos

### Dados móveis:

**SMS:** Os SMS vão “de carona” com os bits trocados entre os celulares e as torres de comunicação.

**MMS:** É a transmissão de mensagens multimídia (áudio, vídeo) por meio de uma espécie de conexão de dados primitiva.

**Conexões móveis:** 1G(10Kbps), 2G(97Kbps), GPRS(32-80Kbps), EDGE(ou 2.75g, 128-236Kbps), 3G(7Mbps), 4G (22,1Mbps), 5g(10Gbps).

### Wi-fi:

**IEEE 802.11:** Define os padrões para wi-fi. o primeiro (IEEE 802.11a) operava a 2Mbps com 2,4 GHz. Hoje em dia, os mais utilizados são IEEE 802.11g e IEEE 802.11n, que transmitem até 54Mbps(g) e 150-600Mbps(n)

**Segurança:** Para segurança, vemos a utilização de:

- WEP: chaves de 64 bits e de 128 bits
- WPA: chaves trocadas periodicamente
- WPA2 (AES) (802.11i): oferece mais segurança, requerendo mais processamento. Ele é um pouco mais lento.

Alguns dispositivos já se conectam diretamente à rede, como: Impressoras, Scanners, e Chromecast.

### Bluetooth:

Realiza uma conexão ponto-a-ponto, não precisando da internet para funcionar.

Possui três classes:

- **1:** possui potência máxima de 100mW e um alcance de até 100m
- **2:** possui potência máxima de 2.5mW e um alcance de até 10m
- **3:** possui potência máxima de 1 mW e um alcance de até 1m



Há um total de **cinco versões**:

Versão	Taxa de transmissão
1.2	1Mbps
2.0 + EDR	3Mbps
3.0	24Mbps
4.0	25Mbps
5.0	50Mbps

Browser, sites, aplicativos e webserver

### Browser:

Conhecido como navegador, é utilizado para acessar sites na internet, identificando diversas LP (linguagens de programação), linguagens de marcação e conteúdo multimídia.

Ademais, os browsers também utilizam plugins, addons e etc. que ajudam a navegação.

Quando se entra num site, ele deixará algumas coisas pré-gravadas em seu computador para facilitar o carregamento. Isto é conhecido como o cache.

### Sites, aplicativos e e-commerce:

#### Sites:

- São páginas da internet
- Servem a diversos propósitos
- Podem ser feitos em diversas linguagens de programação
- D/HTML estão caindo em desuso

#### Aplicativo:

- É um software que é executado num navegador
- Um app de celular, muitas vezes é uma espécie de navegador
- Hoje em dia, já quase não existe diferença entre apps e sites, e os sites já estão em declínio
- Outra diferença que está sumindo é entre programa/software e aplicativo

#### E-commerce:

- É um comércio eletrônico
- São sites de compra e venda com sistemas de pagamento
- Os sistemas de pagamento podem, no entanto, serem exteriores aos sites: Picpay, Boleto, PagSeguro, Paypal, Mercado Pago, etc.

### Web Server:

É onde ficam hospedadas nossas informações e dados.

- Existem dois tipos de web server: estático e dinâmico
- **Estático:** servidor físico onde são armazenados arquivos, softwares e/ou banco de dados
- **Dinâmico:** são os softwares presentes no servidor físico, ex: File Server (arquivos), Application Server (aplicações), Database (banco de dados). No entanto, o mais comum é quando todos estes servers estão juntos num só.

**Web-service:**

É uma interface disponível para fazer requisições e consultas em bancos de dados inacessíveis (ex: consultas no banco de dados do Correios)

## O que são stacks

**Stacks:**

São pilhas de tecnologia: conjunto de softwares necessários e suficientes para executar um aplicativo/programa.

É o ambiente tecnológico com todas suas ferramentas e capacidades para interagir com aplicativos(app)/softwares(sw).

**Importância:** Os desenvolvedores precisam saber das limitações e capacidades das ferramentas e ambientes que têm disponíveis

**Front-end, back-end, full-stack:****Front-end:**

- É a “parte da frente”
- Interface, UI, UX
- Presentes nos sites, softwares, apps, web-service, etc.
- Lógica de programação, HTML, CSS, frameworks JavaScript, etc.

**Back-end:**

- É a “retaguarda”
- Relaciona-se bastante com servidores, banco de dados, etc.
- Trabalha até mesmo no “meio-de-campo”, entre interface e banco de dados.
- MySQL, Oracle, PHP, Java, node, etc.

**Full-stack:**

- Trabalha tanto no Front quanto no Back-end, participando em todas as etapas do desenvolvimento.

## LPs (Linguagens de programação) e Termos

**Principais LPs:**

HTML (ling. marcação)	CSS (ling. marcação)	JavaScript (jQuery, AJAX, outras libs)	
PHP	.NET	ASP	Java
Ruby (On Rails)	Python	Perl	C, C#, C++

**Termos comuns:**

404	Erro conhecido na internet: página não-existente.
Algoritmo	Sequência de passos para se executar uma tarefa.
ALT	Texto alternativo
API	Funcionalidade extra que se utiliza num site para executar outras funções
Aplicação	É o software, aplicativo, programa
Back-end	Stack responsável pela “retaguarda”, aquilo que o usuário final não verá
Biblioteca/dll	Conjunto de ferramentas disponíveis para fazer alguma função

<b>Bootstrap</b>	Um dos frameworks JavaScript mais usados hoje em dia
<b>Breakpoints</b>	Pontos-chave, terminologia usada tanto no front-end quanto no back-end
<b>Browser</b>	Navegador.
<b>Bug</b>	Defeito, falha ou erro no código de um programa que provoca seu mau funcionamento.
<b>Cache</b>	Área de memória onde é mantida uma cópia temporária de dados armazenados em um meio de acesso mais lento, com o objetivo de acelerar a recuperação dos dados.
<b>Código</b>	Instruções feitas em uma ou mais linguagens de programação para que determinado aplicativo funcione e seja executado.
<b>Cont. de versão</b>	Softwares que controlam versões do aplicativo.
<b>Cookies</b>	Cookies são pedaços de código que dão a um site uma espécie de memória de curto prazo, permitindo que ele se lembre de pequenos pedaços de sua informação de navegação.
<b>Debug</b>	Processo de encontrar e remover os erros que podem acometer um programa
<b>Deploy</b>	Colocar no ar alguma aplicação que teve seu desenvolvimento concluído.
<b>Design responsivo</b>	O design se adequa a qualquer tipo de tela, independente da proporção e tamanho.
<b>DNS</b>	Tradutor de nome de site para IP e de IP para nome de site.
<b>Documentação</b>	A interna: são comentários realizados no próprio código A externa: o “manual do usuário” a respeito de alguma LP, por exemplo.
<b>Domínio</b>	O endereço do seu site ou loja virtual na Internet, o nome pelo qual você será encontrado.
<b>DPI</b>	Proporção de pontos na tela, refere-se à nitidez da resolução.
<b>Editor de Texto</b>	O editor de texto é utilizado como uma IDE para se programar.
<b>Estrutura de dados</b>	É o ramo da computação que estuda os diversos mecanismos de organização de dados para atender aos diferentes requisitos de processamento.
<b>Favicon</b>	É o pequeno ícone ao lado do nome do site que irá representá-lo.
<b>Fontes</b>	Pode ser tanto as fontes de letra quanto os códigos/arquivos-fonte.
<b>Framework</b>	“Caixa de ferramentas” disponível para desenvolvimento de aplicativos.
<b>Front-end</b>	Stack relacionada diretamente com o que o usuário final experimenta/visualiza.
<b>FTP</b>	Protocolo de comunicação quase exclusivo para troca de arquivos.
<b>Full-Stack</b>	É responsável tanto pelo Front quanto pelo Back-end, participando de todas as etapas do desenvolvimento.
<b>GitHub/SVN/CV</b>	São os três controladores de versão mais famosos.
<b>HTTP(S)</b>	Camada de segurança do protocolo de comunicação mais usado da internet
<b>IP</b>	O número em que cada indivíduo conectado à rede é identificado
<b>Linguagem</b>	Forma de “comunicação”. Linguagem de programação, marcação, etc.

<b>Meta Tags</b>	Tags HTML que ficam no header, com funções específicas.
<b>Método Ágil</b>	Técnicas de desenvolvimento do software que visam velocidade e qualidade.
<b>Mobile</b>	Dispositivos como smartphone, tablet, iPad, etc.
<b>MVC</b>	Modelo de arquitetura que divide-se em Model, View, Controller-
<b>MVP</b>	Produto mínimo viável para se realizar teste/prototipação.
<b>MySQL</b>	Linguagem usada em banco de dados.
<b>Pixel</b>	Pequenos elementos que compõem uma figura.
<b>Resolução</b>	Quantidade de pixels presentes numa tela.
<b>Servidor</b>	computador usado numa rede para proporcionar algum tipo de serviço (como acesso a arquivos ou a periféricos compartilhados) aos demais componentes da rede.
<b>Sistema Operacional (SO)</b>	Windows, Android, MacOS, etc.
<b>Solução</b>	Programa ou conjunto de programas oferecidos ao cliente como entrega.
<b>SSL</b>	Camada de segurança de sites.
<b>UI</b>	User interface- interface do usuário.
<b>UX</b>	User experience- experiência do usuário.
<b>Versão</b>	Versão do software, em que estado atual ele se encontra.
<b>WYSIWYG</b>	What you see is What you get: o que você vê é o que você ganha.

## Construindo a primeira aplicação

### Aula prática:

Para o PHP, precisamos de um servidor onde a página fique hospedada para que ela seja apresentada pelo navegador.

XAMPP é a IDE mais popular de PHP.

O X de XAMPP refere-se ao sistema operacional, X significa que é aplicável em qualquer um dos três principais: Windows, Linux e MacOS

O PHP está escrito nas tags <?php ?>

## INTRODUÇÃO À CRIAÇÃO DE WEBSITES COM HTML5 E CSS3

### Aula prática:

O elemento HTML estrutura-se da seguinte forma: `<h1 class="título">Título</h1>`, onde:

- tag: `<>` para aberta e `</>` para fechada.
- atributo, denominado "título"
- conteúdo, denominado Título

A estrutura básica de um documento html é:

`<!DOCTYPE html>`

`<html>`

`<head>`

```
<meta>
<title></title>
</head>
<body>
</body>
</html>
```

### **<html>**

A tag html é a raiz do seu documento, todos os elementos HTML devem estar dentro dela. E nela nós informamos ao navegador qual é o idioma desse nosso documento, através do atributo lang, para o português brasileiro usamos pt-BR.

### **<head>**

A tag head contém elementos que serão lidos pelo navegador, como os metadados - um exemplo é o charset, que é a codificação de caracteres e a mais comum é a UTF-8, o JavaScript com a tag script, o CSS através das tags style e link - veremos a diferença quando falarmos sobre CSS - e o título da página com a tag title.

### **<body>**

E dentro da tag body colocamos todo o conteúdo visível ao usuário: textos, imagens, vídeos.

### **<h1>-<h6>**

Eles não foram criados na versão 5 do HTML e nem são específicos para semântica, mas servem para esse propósito. São utilizados para marcar a importância dos títulos, sendo <h1> o mais importante e <h6> o menos. Uma dica: use apenas um <h1> por página, pois ele representa o objetivo da sua página.

## Entendendo o que é semântica

### **Semântica:**

A semântica nos permite descrever mais precisamente o nosso conteúdo, agora um bloco de texto não é apenas uma div, agora é um article e tem mais significado assim. E temos vários elementos para ressignificar as divs:

### **<section>**

Representa uma seção genérica de conteúdo quando não houver um elemento mais específico para isso.

### **<header>**

É o cabeçalho da página ou de uma seção da página e normalmente contém logotipos, menus, campos de busca.

### **<article>**

Representa um conteúdo independente e de maior relevância dentro de uma página, como um post de blog, uma notícia em uma barra lateral ou um bloco de comentários. Um article pode conter outros elementos, como header, cabeçalhos, parágrafos e imagens.

## <aside>

É uma seção que engloba conteúdos relacionados ao conteúdo principal, como artigos relacionados, biografia do autor e publicidade. Normalmente são representadas como barras laterais.

## <footer>

Esse elemento representa o rodapé do conteúdo ou de parte dele, pois ele é aceito dentro de vários elementos, como `article` e `section` e até do `body`. Exemplos de conteúdo de um `<footer>` são informações de autor e *links* relacionados.

### Primeiro exercício: montando um template simples de página com posts

Para montar uma simples página com posts, poderemos fazer da seguinte maneira:

```
<!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">
```

## <head>

## <body>

```
<header>
  <!-- Neste primeiro header, fica o cabeçalho da página-->
  <h1>Seja bem-vindo</h1>
</header>
<!-- Botaremos, nesta section, o conteúdo dela-->
<section>
  <!-- Este segundo header terá o título do conteúdo geral-->
  <header>
    Meus posts
  </header>
  <!-- Já aqui teremos o conteúdo específico, num article (ele
irá ocupar boa parte da pag.)-->
  <article>
    <!-- Este terceiro header será o título deste conteúdo
específico: o nosso post-->
    <header>
      <h3>Post #1</h3>
    </header>
    <!-- Logo abaixo, teremos o conteúdo-->
```

```
        bla bla bla bla
    </article>

    <!-- Finalizando o article, no final da página, temos o rodapé
    com informações adicionais-->

    <footer>
        Informações adicionais
    </footer>

</section>
```

<body>

Após este primeiro exercício, espera-se que o site tenha essa cara:

## Seja bem-vindo

Meus posts

### Post #1

bla bla bla bla

Informações adicionais

## Como usar textos de links em html

### Tags para texto:

Podemos hierarquizar o destaque de cada texto utilizando as tags <hx></hx>, podendo o 'x' variar de 1 a 6.

Usualmente, utilizamos da seguinte maneira:

- <h1></h1>: Título de página
- <h2></h2>: Título de seção
- <h3></h3>: Título de artigo
- Também utilizamos o <p></p>, que representa parágrafos, compreendendo não só textos, como vídeos, código e afins.

Podemos gerar automaticamente um lorem ipsum no conteúdo, para ver como o texto ficará. Só precisamos escrever **loremx**, sendo 'x' o número de palavras que você almeja ver.

### Tags para links:

Outro elemento bastante importante é a âncora, representado por <a></a>, utilizado para interligar diversos sites.

Podemos fazê-lo abrir uma página ou uma aba nova. Para abrir uma página diretamente sobre a que estávamos vendo, utilizamos a seguinte sintaxe:

```
<a href="https://www.linkedin.com/in/alexandre-psouza">Linkedin</a>
```

Para abrir uma nova aba ao invés de sair da página em que estávamos, utilizaremos a **target="\_blank" após o link**. com isto, a sintaxe ficará da seguinte maneira:

```
<a href="https://www.linkedin.com/in/alexandre-psouza"
target="_blank">Linkedin</a>
```

**NÃO ESQUECER DO HTTPS:// NA FRENTE, CASO CONTRÁRIO A PÁGINA NÃO IRÁ FUNCIONAR.**

Temos a função de enviar diretamente um email para algum endereço predeterminado, através do seguinte comando:

```
<a href="mailto:alexandre@gmail.com">E-mail</a>
```

Como fica o exercício anterior se adicionarmos a ele links, e-mail e texto?

```
<!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>TITULO</title>
</head>
<body>
  <header>
    <h1>Seja bem-vindo</h1>
  </header>
  <section>
    <header>
      Meus posts
    </header>
    <article>
      <header>
        <h3>Post #1</h3>
      </header>
      <!-- Adicionamos o texto abaixo, compreendido por <p></p> para
indicar parágrafo, e preenchido com lorem ipsum, feito com o comando
lorem20, indicando que quero 20 palavras.-->
      <p>
        Lorem ipsum, dolor sit amet consectetur adipisicing
elit. Itaque recusandae quasi dolor deserunt fugiat! Ad eligendi
quibusdam officiis deserunt consequuntur dicta eos dolorem magni
perferendis.
      </p>
    </article>
    <!-- Ao rodapé, adicionamos E-mail e LinkedIn. O mailto irá
abrir diretamente sua plataforma de email, preparando um envio para
```



```
alexandre@gmail.com, a ancora associada ao linkedin termina com
target="blank", indicando que quero abrir o link numa nova aba-->
<footer>
    <a href="mailto:alexandre@gmail.com">E-mail</a>
    <a href="https://www.linkedin.com/in/alexandre-psouza"
target="_blank">Linkedin</a>
</footer>
</section>
</body>
</html>
```

Depois desta segunda parte, nosso site terá esta cara:

## Seja bem-vindo

Meus posts

Post #1

Lorem ipsum, dolor sit amet consectetur adipisicing elit. Itaque recusandae quasi dolor deserunt fugiat! Ad eligendi quibusdam officiis deserunt consequuntur dicta eos dolorem magni perferendis.

[E-mail](#) [Linkedin](#)

## Como usar textos de links em html

### Tag img:

A tag <img> irá conter uma imagem que você desejar. Teremos dois elementos:

```

```

O **src** (source) é obrigatório. Ele guarda o caminho de uma imagem que possa estar em seu diretório ou em algum outro lugar.

```
<img alt="Minha foto">
```

O **alt**, mesmo não sendo obrigatório, é essencial para questões de acessibilidade. Ele mostra a descrição da foto quando ela não é carregada. Leitores de tela usam a descrição dada para informar aos indivíduos o que a imagem significa.

Um exemplo da aplicação destes dois elementos numa imagem é o seguinte:

```

```

Se eu adicionar, por exemplo, ao meu header, uma foto minha, este será o código:

```
<header>
    <!-- Neste primeiro header, fica o cabeçalho da página-->
    
    <h1>Seja bem-vindo</h1>
</header>
```

Posso também adicionar uma foto ao header do post que eu fiz:

```
<header>

    <h3>Post #1</h3>

</header>
```

Desta forma, meu site ficará assim:



## Seja bem-vindo

Meus posts

### Post #1



Lorem ipsum, dolor sit amet consectetur adipiscing elit. Itaque recusandae quasi dolor deserunt fugiat! Ad eligendi quibusdam officiis deserunt consequuntur dicta eos dolorem magni perferendis.

[E-mail](#) [Linkedin](#)

## Como organizar listas com HTML

### Tags li, ul e ol:

Cada uma das tags citadas acima irá fazer uma lista com características diferentes:

- **<ul></ul>** representa uma lista em que a ordem dos elementos **não importa**  
**ex:**  
Item X  
Item Y
- **<ol></ol>** representará uma lista em que a ordem dos itens **é importante**  
**ex:**  
1. Item X  
2. Item Y
- **<li></li>** representa um item dessa lista, como o próprio Item X e Y, por exemplo

No rodapé de meu código, organizei meus contatos numa lista não-ordenada (ul). O resultado final deste HTML foi o seguinte:

```
<!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>TITULO</title>
```

```

</head>
<body>
    <header>
        
        <h1>Seja bem-vindo</h1>
    </header>
    <section>
        <header>
            Meus posts
        </header>
        <article>
            <header>
                <h3>Post #1</h3>
                
            </header>
            <p>
                Lorem ipsum, dolor sit amet consectetur adipisicing
elit. Itaque recusandae quasi dolor deserunt fugiat! Ad eligendi
quibusdam officiis deserunt consequuntur dicta eos dolorem magni
preferendis.
            </p>
        </article>
        <footer>
            <!-- Aqui adicionei minha UL. Não esquecer de usar o <li></li> em
cada um dos elementos da lista-->
            <ul>
                <li>
                    <a href="mailto:alexandre@gmail.com">E-mail</a>
                </li>
                <li>
                    <a
href="https://www.linkedin.com/in/alexandre-psouza"
target="_blank">Linkedin</a>
                </li>
            </ul>
        </footer>
    </section>
</body>
</html>

```

No final, esta foi a cara do site:



## Seja bem-vindo

Meus posts

Post #1



Lorem ipsum, dolor sit amet consectetur adipisicing elit. Itaque recusandae quasi dolor deserunt fugiat! Ad eligendi quibusdam officiis deserunt consequuntur dicta eos dolorem magni perferendis.

- [E-mail](#)
- [LinkedIn](#)

## Introdução aos conceitos básicos do CSS3

### Introdução ao CSS3:

A sintaxe é simples: cria-se regra para elementos ou grupos deles.

**a, h1** {color:#ffff; font-size:15px}

- São os seletores as tags, classes e ids definidas no html
- Nas declarações, faremos as alterações que queremos

Os ids e classes são responsáveis por conferir uma individualidade a alguma tag, para que não se mude todas as tags de uma vez e fiquem todas iguais.

No css, nos referimos às classes com **.** e aos ids com **#**.

O id só pode ser utilizado uma vez na página, enquanto a class é repetível.

Para incorporar um CSS a um html, inserimos no <head> o seguinte comando:

```
<link rel="stylesheet" href="style.css">
```

(o style.css é o nome do arquivo que eu criei na pasta)

Um exemplo da aplicação do CSS:

Após atribuir uma id/classe aos h1, h2 e h3, na página do css, fiz o seguinte:

```
#title, .subtitle, .post_title {  
    color: blue;  
    font-weight: bold;  
}
```

Após isso, os títulos ficaram em azul.

### Conceitos básicos:

- **Background:** O background é o fundo do elemento em questão. Pode ser cor, fotos, etc.

ex:

```
background-color: #fff;
```

Recomenda-se utilizar o código hexadecimal para definir as cores, tendo em vista que é mais fácil de encontrar materiais e fontes sobre ele.

Para colocarmos uma imagem de fundo no background, fazemos o seguinte:

```
background-image: url("bg.png");
```

- **Margin:** A margem será o espaçamento ao redor do elemento, considera-se de seu limite para fora

ex:

```
margin: 18px;
```

- **Border:** As bordas são os limites do elemento.

ex:

```
border: 3px solid black;
```

**solid:** mostra uma borda simples e reta;

**dotted:** são bolinhas com um pequeno espaçamento entre elas;

**dashed:** forma uma linha tracejada.

**Border-radius:** Permite arredondar os cantos de um elemento.

As unidades mais comuns são os pixels e as porcentagens. Se utilizarmos apenas um valor, ele irá mudar uniformemente todos os lados, mas podemos controlar isso semelhante a como controlamos os paddings/margins, exemplificado abaixo no próximo tópico.

- **Padding:** Padding é o espaçamento entre a borda e o interior do elemento, enquanto a margem pega “de dentro para fora”, o padding vai “de fora pra dentro”

ex:

```
padding: 18px;
```

Há três formas de definirmos os padding/margin:

se botarmos apenas um número, conforme o exemplo acima, haverá um padding uniforme entre todas as direções,

Caso ponhamos dois números, será considerado que para o eixo vertical (superior e inferior), haverá 18px de padding, e para o horizontal (esquerda/direita) haverá 10px.

```
padding: 18px 10px;
```

Se botarmos 4 números, será lido da seguinte forma: **o primeiro valor será o topo, o segundo a direita, o terceiro a parte inferior e o quarto a esquerda.**

```
padding: 18px 10px 5px 0;
```

Podemos também definir manualmente como será cada lado:

```
background-color: #fff;
padding-top: 5px;
padding-bottom: 5px;
padding-right: 5px;
padding-left: 5px;
```

Às vezes, o próprio navegador já adiciona automaticamente margens. Para ajustarmos isso, devemos definir que a margin seja 0 onde notarmos este problema, e defini-las manualmente.

### Estilizando textos:

- **font-family:** altera a fonte dos textos, pode ser tanto da web quanto da máquina. vale se atentar para as web-safe fonts. Aquelas que estão na maioria das máquinas e dispositivos. (ex: Verdana e Arial)  
podemos adicionar mais de uma fonte, para que caso a primeira não funcione, a segunda sirva como uma segunda opção
- **font-size:** define o tamanho do texto, pode ser ditado em pixels.
- **font-style:** muda a aparência do texto. Por exemplo, é no font style que definiremos um texto como itálico. (devemos ver se a fonte suporta o itálico)
- **font-weight:** define o peso do texto. É semelhante ao nosso negrito daqui do word.
- **text-transform:** alterna o texto entre maiúsculos e minúsculos, temos o **uppercase** para deixar tudo em maiúsculo, **lowercase** para deixar tudo em minúsculo e **capitalize** para deixar tudo começando em maiúsculo.
- **text-decoration:** é utilizado para dar destaque ao contexto, utilizando-se de linhas, underlines e overlines.

### Estilizando listas:

Podemos alterar os marcadores de nossa lista, além de retirá-los por completo.

Para retirá-los, usamos o:

```
list-style: none;
```

Para adicionarmos às listas outras características, usamos **list-style-type**, ex:

- **list-style-type: square;** transforma os marcadores em quadrados
- **list-style-type: upper-roman;** transforma marcadores em números romanos em letras maiúsculas
- **list-style-type: "1F44D";** transforma marcadores em joinha.
- **list-style-image: url("xxx.jpg");** adiciona imagem como marcadores.

## IMPORTANTE: PODEMOS NOS REFERENCIAR AOS ELEMENTOS LISTANDO A ORDEM EM QUE ESTÃO INSERIDAS!

por exemplo:

```
ul li a {  
    color: navy;  
}
```

O a estava dentro do li que estava dentro do ul.  
Isto permite um maior controle do que iremos ajustar.

## Propriedade de dimensões e alinhamento

### Dimensão e alinhamento:

- **Width e Height:** é a largura e altura do elemento. Podemos definir, por exemplo, que ele ocupará 50% do elemento em que ele se insere, que ele sempre irá ocupar tal proporção
- **Max-width e Max-height:** são as larguras e alturas máximas do elemento, a proporção ou medida-limite que eles poderão chegar.
- **Margin:** cria espaçamento entre os elementos, e seu valor **auto** pode criar tais espaçamentos automaticamente
- **Text-align:** alinha o texto dentro de um conteúdo






## IDE - INSTALAÇÃO E CONFIGURAÇÃO


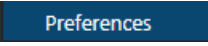

### VS Code- Recursos básicos

#### Visual Studio Code:

É a IDE mais popular do desenvolvimento web.

Elementos do VS Code:

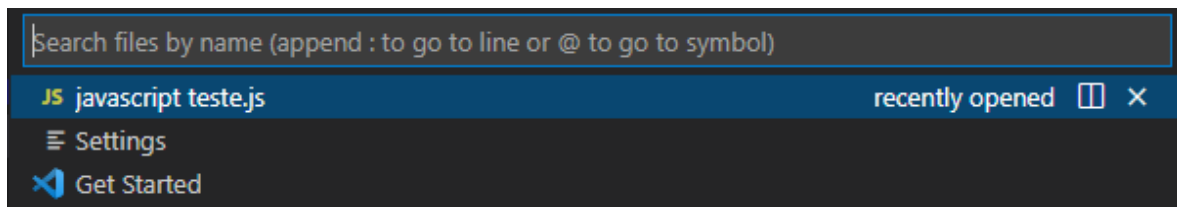
-  O **Explorer** permite você navegar pelos arquivos, seja os que já estão abertos, seja os que estão presentes numa pasta selecionada.
-  O **Search** permite que você busque, por exemplo, uma palavra-chave dentro de todos os seus arquivos
-  O **Source Control** integra o VS Code com a ferramenta de controle de versões (ex:Github). Você pode iniciar um repositório, clonar repositório, e até mesmo publicar no Github.
-  O **Run & Debug** irá rodar um código selecionado, permitindo que o programador analise-o e possa debugá-lo com maior facilidade.
-  As **Extensions** são maneiras de tornar o VS Code uma ferramenta ainda mais potente do que ela já é.

Caso queiramos mudar alguma preferência/opção, podemos vir em **File**  e buscar por **Preferences** . Após isso, selecionamos **Settings**  para ir às opções (atalho: ctrl+,).

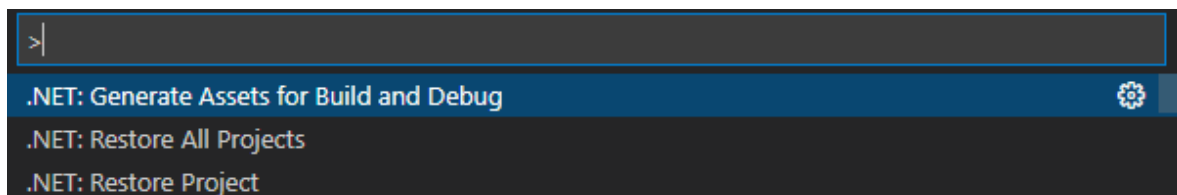
Exemplos de alterações são: tamanho da fonte, distanciamento da tab, autosave, etc...


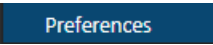
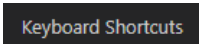
Algo legal de se ativar é o **word wrap**, para que uma linha não tenha uma distância infinita e todo seu conteúdo fique na tela.

Um atalho bem útil: se usarmos o **ctrl+p** no VS Code, iniciamos a **busca de arquivos**:



Outro atalho interessante é o **ctrl+shift+p**, que nos mostra uma série de comandos.




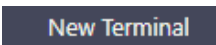
Se quisermos ver os atalhos e ajustá-los, podemos ir novamente em **File** , buscar por **Preferences**  e ir em **Keyboard Shortcuts**  (atalho: ctrl+K+S).

Vale citar o **Multicursor Modifier**, que está nas **Settings**. Com ele, ao segurarmos a tecla **alt** podemos alterar vários espaços/elementos simultaneamente.

Uma denominação que vale a pena ser citada é o **emmet**: estrutura base de uma linguagem, que pode ser posta para adiantar os processos. um exemplo é o comando `html:5` no VSCode, que já nos gera a seguinte estrutura:

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

### Terminal e Git:

Para criar um terminal, vamos em **Terminal** , no menu superior e selecionamos **New Terminal** . (atalho: ctrl+shift+'). Qualquer comando que faríamos num terminal do computador poderá ser realizado aqui, no terminal do VSCode.

## INTRODUÇÃO AO JAVASCRIPT

### O que é o JavaScript?

#### História, evolução e aplicações:

Ela é uma linguagem **interpretada** (ou seja, o browser está rodando ela em tempo real), ou seja, passa por menos processos de interpretação do computador.

JavaScript é uma **linguagem multiparadigma**, (ex: pode ser usada em programação orientada a objetos, programação funcional, etc.).

Também vale citar o fato do JavaScript ser comumente utilizada em aplicações web client-side, e segue o padrão **ECMAScript**.

#### Algumas aplicações do JavaScript:

- Web
- Mobile
- Smartwatches
- Games
- Internet of Things
- APIS



### Manipulando um arquivo:

- **Comentários:**

Comentários serão elementos que não vão ser lidos pelo código.

Fazer um comentário é simples. Para uma linha, usamos `//`:

```
1 // ISTO É UM COMENTÁRIO
```

Para fazermos um comentário de várias linhas, começamos om `/*` e acabamos com `*/`:

```
1 /* ISTO É UM COMENTÁRIO
2 DE DUAS LINHAS, POR EXEMPLO */
```

Se selecionarmos uma linha e digitarmos **ctrl+K+C**, criamos um comentário

- **Variável (var) e constante (const):**

Conforme citado no primeiro capítulo:

A variável pode assumir qualquer um dos valores de um determinado conjunto de valores.

Regras das variáveis:

Atribuição de um ou mais caracteres

Primeira letra - não número

Sem espaços em branco

Utilização de palavras reservadas

Caracteres e números

Uma variável que é inalterável é uma **constante (const)**, como, por exemplo, `pi=3,14`

No geral, as constantes são escritas em letras maiúsculas, ex: `(const PRECO=2)`

- **Funções (function)**

Funções são formatadas da seguinte maneira:

```
1 function soma() {
2
3 }
```

nos `()`, declaramos os atributos/parâmetros da função.

Neste seguinte exemplo, significa que quero dois elementos, o `a` e o `b`.

```
1 function soma(a,b) {
```

Para realizar, de fato, uma soma destes dois elementos, botaremos o comando dentro dos `{}`.

Pediremos para a função retornar o valor de `a+b`

```
function soma(a,b) {
  return a+b
}
```

podemos mostrar o resultado desta soma no **console**, com o comando `console.log(a+b)`

```
1 function soma(a,b) {
2   console.log(a+b)
3 }
```

### Quando utilizá-los?

- **return:** geralmente quando iremos utilizar o resultado desta função dentro de outra.
- **console.log:** quando queremos ver o valor que a função está dando, para geralmente chegar o andamento, debugar, etc.

Para chamarmos uma função, deveremos botar seu nome e, dentro de seus parênteses, as variáveis que equivalem aos elementos dela. (no caso do exemplo acima, **uma equivalente ao a** e outra **equivalente ao b**)

```
soma(3,4);
```

O 3 equivaleria ao **a** de minha função, e o 4 equivaleria ao **b**.

## Console

### Executando um arquivo .js:

Podemos usar o console tanto dentro de uma página web quanto num terminal.

- **Na Web:**  
Clicando com o botão direito na tela e indo em **Inspecionar Elemento**, abriremos o console. Também poderemos utilizar os atalhos **ctrl+shift+i** ou **F12**.  
Em muitos casos de client-side, poderemos fazer alterações no próprio terminal web.
- **No console:**  
No VSCode, se abrirmos um terminal com o bash, (com o **Node.js** já instalado) e fomos no diretório onde está nosso arquivo .js, podemos usar o comando **node nome do arquivo.js** para executá-lo em nosso log do terminal.

**NÃO ESQUECER DE INSTALAR O NODE.JS!**

## Javascript em uma página da web

### Estrutura de projetos:

Para um projeto, teremos o nosso arquivo-base como um **html**, geralmente indicado como **index**, e alguns arquivos que o apoiarão (dentro de outra pasta, para ficar mais organizado).

```
> assets ●
<> index.html U
```

Neste caso, criamos a **assets**, e dentro dela botamos os arquivos **css** e **javascript**

```
▼ assets ●
  ▼ css ●
    # styles.css U
  ▼ js ●
    JS script.js
  <> index.html U
```

### Inserindo javascript numa página HTML:

Para inserirmos um script num html, usamos o seguinte comando:

```
<script src="assets/js/script.js"></script>
```

Diferente das referências aos arquivos css, que se auto-fecham, precisamos fechar a referência aos javascripts com outro </script>

```
<link rel="stylesheet" href="assets/css/styles.css"/>
<script src="assets/js/script.js"></script>
```

Se abrirmos o html em nosso navegador e o inspecionarmos, poderemos ver que suas informações estão condizentes.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <link rel="stylesheet" href="assets/css/styles.css">
    <script src="assets/js/script.js"></script>
    <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>Teste</title>
  </head>
```

### Interagindo com elementos DOM:

DOM significa Document Object Model, relaciona-se com a estrutura dos elementos dentro da janela. Forma-se uma árvore de dependência entre os elementos criados onde podem ser irmãos, filhos ou pais.

Se viermos no console do navegador e inserirmos o comando abaixo, ele irá nos mostrar os elementos que têm relação com h1.

```
> document.getElementsByTagName ('h1')
< HTMLCollection [h1]
```

Ao confirmarmos, o console nos retorna que há uma lista de apenas 1 elemento inserido no h1, e sua posição nesta lista é a 0.

```
> document.getElementsByTagName ('h1')
< ▼ HTMLCollection [h1] ⓘ
  ▶ 0: h1
    length: 1
  ▶ [[Prototype]]: HTMLCollection
```

Sabendo que o elemento está na posição 0 desta lista, eu posso criar uma variável que a referencie:

**(CRIAMOS A VARIÁVEL PARA AGILIZAR OS PROCESSOS, EVITANDO QUE TODA VEZ TENHAMOS QUE ESCREVER document.getElementsByTagName('h1')[0])**

```
> var heading1= document.getElementsByTagName('h1')[0]
< undefined
> heading1
< <h1>Olá a todos</h1>
```

Ao chamar a variável, ela irá me indicar qual é o elemento que está na posição 0 da lista: o "<h1>Olá a todos</h1>"

Podemos mudar dentro de nosso próprio navegador a cor do elemento para vermelho, por exemplo:

```
> heading1.style.color='red'  
< 'red'
```

**Olá a todos**

## Mercado de Trabalho

### Frameworks:

Exemplos: VueJS, Angular, React e JQuery

São ferramentas desenvolvidas por pessoas/empresas para facilitar o desenvolvimento em JavaScript.

Embora no mercado de trabalho geralmente não se cobre JavaScript puro, é essencial aprendermos ele antes de partirmos para algum framework.

### Mercado de Trabalho:

Sites importantes:

- **W3C**
- **MDN web docs**
- **Github**
- **StackOverflow**
- **Linkedin**
- **Youtube**

**Disclaimer:** O tópico de Git/Github está aqui, e não no começo, porque estou seguindo a minha ordem de estudo. Eu preferi ver uma introdução de tudo que vamos abordar antes: JavaScript, Visual Studio Code, etc. para só depois ir ao Git/Github, que será onde vamos adicionar nossos arquivos.

## GIT E GITHUB

### Introdução ao Git e sua importância

#### O que é o GIT?:

É um sistema de versionamento de código criado em 2005.

GIT E Github são duas tecnologias diferentes, mas complementares.

Arelado a eles, temos uma série de vantagens:

- Controle de versão
- Armazenamento em nuvem
- Trabalho em equipe
- Possibilidade de melhoramento de código
- Reconhecimento

### Navegação via command line interface e instalação

#### Comandos básicos para um bom desempenho no terminal:

O Git é um CLI (command line interface), ou seja. Ele não possui uma interface gráfica e opera por linhas de comando.

alguns comandos básicos que são essenciais:

- **Listar as pastas**

No Git Bash, utilizaremos o comando **\$ ls** para listar

```
Alexandre@DESKTOP-22RMIOE MINGW64 ~  
$ ls  
'3D Objects'/  
'Ambiente de Impressão'@  
'Ambiente de Rede'@  
AndroidStudioProjects/  
AppData/
```

(caso estejamos no terminal windows, o comando é **dir**)

- **Mudar de pastas**

Para mudar de pastas, usamos o comando **\$ cd**, que significa *change directory*. se quisermos retroceder para outro nível, podemos usar **\$ cd ..** para retornar um nível.

(o windows utiliza o mesmo comando)

- **Limpar o terminal**

Se usarmos o **\$ clear**, o terminal será limpo, evitando poluição.

No bash, temos o atalho **ctrl + L**

(para o terminal windows, temos o **>cls**, que significa clear screen)

- **Autocompletar sentenças**

Tanto no windows quanto no bash, se usarmos a tecla **tab**, as sentenças serão completadas automaticamente.

Se usarmos a seta para cima, o terminal autocompletará comandos que já utilizamos anteriormente.

- **Criar as pastas/arquivos**

Para criarmos arquivos, usamos o comando **mkdir**, tanto no windows quanto no bash. significa *make directory*.

```
Alexandre@DESKTOP-22RMIOE MINGW64 /c  
$ mkdir workspace
```

para criar um novo arquivo, pode-se usar o comando **echo** (que retornará algo que você escreveu) associado a algum nome. este nome será um arquivo que será criado:

```
Alexandre@DESKTOP-22RMIOE MINGW64 /c/workspace  
$ echo hello > hello.txt
```

- **Deletar pastas/arquivos**

No windows, ao utilizarmos o comando **del *nomedoarquivo***, ele irá deletar tudo que está dentro do arquivo, mas não a pasta dele propriamente dita.

Para deletarmos completamente uma pasta do windows, usamos o comando **rmdir *nomedoarquivo*** (que significa *remove directory*).

Para o Bash, usamos o comando **rm** (de remove), podendo adicionar flags que reforcem algumas coisas de nossa remoção. Se utilizarmos **rm -rf**, a flag **r** significará “recursivo”, significando que, caso haja alguma pasta dentro deste diretório, ela também será excluída. Já o **f** significa “force”, ou seja, não nos aparecerá nenhuma solicitação de confirmação.

Entendendo como o Git funciona por debaixo dos panos

### Tópicos essenciais para entender o funcionamento do Git:

- **SHA1**

Secure Hash Algorithm: um algoritmo de encriptação projetada pela NSA (agência de segurança nacional dos EUA)

Gera um conjunto de caracteres de 40 dígitos único.

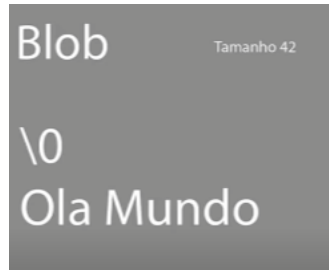
```
1 echo "ola mundo" | openssl sha1  
2 > (stdin)= f9fc856e559b950175f2b7cd7dad61facbe58e7b
```

para checarmos qual o algoritmo de um arquivo, usamos o comando **openssl sha1 *nomedoarquivo***

### Objetos internos do Git:

#### Blobs:

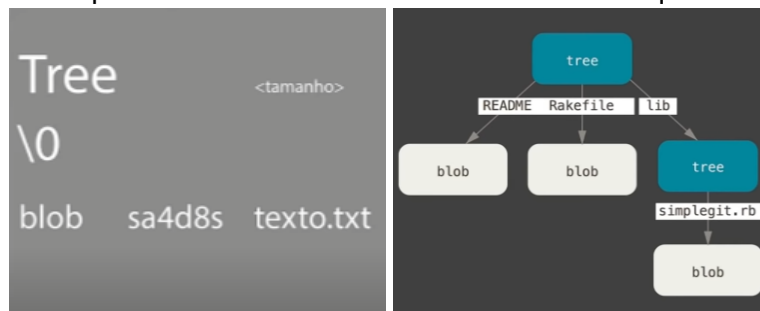
Os arquivos ficam dentro de objetos chamados **blobs**, que contêm metadados dentro de si. dentro dele haveremos o tipo do arquivo, o tamanho, uma \0 e o conteúdo propriamente dito.



#### Trees:

Elas armazenam os blobs. Estão num nível de complexidade acima dos blobs, que são as estruturas mais básicas.

São responsáveis por montar a estrutura de onde estão os arquivos.



#### Commit:

É o que irá juntar tudo, ele apontará para uma árvore, parente, autor e afins. Além de também ter uma timestamp e poder conter uma mensagem.



### Chave SSH e Token:

O processo de autenticação por apenas senha está se tornando obsoleto. Outras alternativas surgem para melhorar a segurança:

- **Chave SSH:**

É uma forma de estabelecer uma conexão segura e encriptada entre duas máquinas.

Para se gerar uma chave SSH:

1. Abrimos o Git Bash
2. escrevemos: `ssh-keygen -t ed25519 -C seuemail`
3. criamos a senha
4. confirmamos a senha

5. navegamos para a pasta: no meu caso foi `c/users/Nome/.ssh`
  6. listamos os comandos com **ls**
  7. usamos **cat id\_ed25519.pub**
  8. copiamos o resultado
  9. vamos ao Github, em SSH and GPG Keys e adicionamos uma nova chave SSH
  10. inserimos a chave SSH que copiamos no passo 8, botamos um título e confirmamos
  11. Após isso, dentro da pasta `.ssh`, rodamos o comando **eval \$(ssh-agent -s)**
  12. O comando nos dará um Agent pid aleatório
  13. adicionamos o agent à chave privada com **ssh-add id\_ed25519**
  14. inserimos nossa senha e confirmamos
- **Token de acesso pessoal:**

Os tokens de acesso são mais semelhantes às formas de login tradicionais. No Github geramos um token que deve ser memorizado pelo autor em algum lugar. Pode-se estabelecer uma data de “validade” para ele, sendo necessário após sua expiração gerar outro.

## Primeiros comandos com Git

### Iniciando um Git e criando um commit:

- **git init**

Através do git init, iniciamos o repositório do git
- **git add**

Usamos o git add para adicionar arquivos ao nosso estágio de stage, salvando as alterações e os deixando prontos para commit
- **git commit**

Para “fechar” um git, usamos o git commit

  1. Ao abrirmos o diretório onde queremos trabalhar (de acordo com sua preferência), podemos iniciar o Git Bash e entrar com o **git init**.
  2. Isto irá criar uma pasta oculta: `.git`, para visualizarmos arquivos ocultos usamos **ls -a**
  3. Com o git inicializado, dentro desta pasta master nós usaremos os seguintes comandos:
  4. **git config --global user.email "seu\_email"**
  5. **git config --global user.name seunick**
  6. Quando finalizarmos nossas alterações, usamos **git add \***
  7. e submetemos com **git commit -m "nome do commit"**
- **git push**

O git push é responsável por enviar o nosso commit para um repositório remoto. Utilizamos quando vamos mandar um arquivo para o github  
`git push origin main`
- **git pull**

É o contrário do git push. Aqui nós pegamos os arquivos do repositório remoto.
- **git clone**

Após copiarmos a url de um repositório remoto, podemos cloná-lo em nossa máquina usando o git clone.

Para checarmos configurações gerais a respeito do nosso git (até mesmo quais minhas credenciais são) usamos o **config – list**

Caso precisemos retirar o username e alterá-lo, usaremos o comando: **git config – global – unset user.email** e repetimos o processo anterior

- **copiar**

para copiar, usamos o **cp**. Pegamos a localização da pasta/arquivo que queremos copiar, e ao lado botamos a localização de onde vamos copiar, ficando, por

exemplo, assim

```
cp -r /c/workspace/livro-receitas /c/workspace/resumo-decola-tech-3
```

o **-r** foi usado para que possamos copiar todos os arquivos que estejam dentro da pasta selecionada.

- **mover**

Usamos o **git mv** para mover uma pasta ou arquivo de um canto para outro. ao fazermos isso, podemos até renomeá-las, se movermos um arquivo para outro com nome diferente, por exemplo:

```
git mv fonte destino
```

## Ciclo de vida dos arquivos Git

### Passo a passo do ciclo de vida:



Para monitorarmos os **status** de nossos arquivos Git, usamos o comando **git status**. No nosso ambiente de desenvolvimento, teremos o diretório de trabalho (**working directory**), a staging area e o repositório local. Para podermos enviar nosso trabalho a um repositório remoto, precisaremos executar uma série de comandos.

- **Aplicar a tudo**

Caso queiramos, por exemplo, fazer um **git add** a todos os arquivos, basta utilizar um **\***

- **Submeter ao github:**

Depois de criar um repositório remoto no Github, vamos ao nosso bash, com a url copiada, e usamos o seguinte comando: **git remote add origin *url da origem***

## SINTAXE BÁSICA EM JAVASCRIPT

### O que danado é o JavaScript?

#### Exemplo de utilização do JavaScript:

Criando um arquivo html básico, podemos inserir este comando em seu body:

```
<script>
  alert("Bem-vindo!")
</script>
```

Ao fazermos isto e abrirmos o arquivo html, o seguinte nos aparecerá:



127.0.0.1:5500 diz

Bem-vindo!

OK

Uma boa prática a se fazer, no entanto, é externarmos o arquivo script, carregando-o ao final do body.

```
<script src="assets/js/script.js"></script>
</body>
```

## Entendendo variáveis e seus valores

### Entendendo variáveis e seus valores:

O Javascript já realiza automaticamente a conversão de tipos de variáveis. Se nós escrevermos alguma envolvida por "", ele já entenderá que estamos nos referindo a uma String.

Para checarmos os tipos da variável, usamos **console.log(typeof(variavel));**

Variáveis podem guardar valores dos tipos:

- **Boolean**  
São variáveis que podem ser **true** ou **false**
- **null**  
São variáveis que não possuem nenhum valor
- **undefined**
- **Number**  
Como o nome já diz, são os **números**
- **String**  
As strings equivalem a **textos**
- **Array**
- **Object**
- **Function**  
São as funções que serão evocadas para realizar alguma ação.

### Declaração de variáveis

Existem 3 modos de declarar as variáveis em JavaScript:

- **var**  
Escopo global e local, pode ter seu valor alterado, se não tiver um valor inicial será tratada como null
- **let**  
Escopo local de bloco, pode ter seu valor alterado, se não tiver um valor inicial será tratada como null;
- **const**  
Escopo local de bloco, somente leitura, o valor inicial é obrigatório e não pode ser alterado

### Escopo

Definirá a limitação e a visibilidade de um bloco de código.

- **Escopo global**  
Quando a variável é declarada fora de qualquer bloco, sua visibilidade fica disponível em todo o código
- **Escopo local**  
Quando a variável é declarada dentro de um bloco, sua visibilidade pode ficar disponível ou não  
Quando quisermos saber o tipo de uma variável let, que estará dentro de uma

função, por exemplo, devemos inserir o comando **console.log(typeof(variavel));** dentro da função.

```
function escopoLocal() {  
  let escopoLocalInterno = 'local';  
  console.log(escopoLocalInterno)  
}
```

### Regra de uso de variáveis

- Iniciar com letras, \_ ou \$, nunca iniciar com números
- Não usar espaços. Usar camelCase ou \_
- Não usar palavras reservadas da linguagem
- Declarar variáveis no topo de bloco de código

### Atribuição =

usando apenas 1 '=', cria-se uma atribuição. Um valor, com isso, é atribuído a alguma variável.

```
var atribuicao = 'meunome'
```

### Comparação ==

Para comparar, usamos '='. O valor retornado será um booleano de true ou false, para nos indicar se um valor é igual a outro.

ex:

```
var comparacao = '0' == 0
```

O console nos retornará que a afirmação é verdadeira.

### Comparação Idêntica ===

A comparação idêntica, através de '===' irá considerar, além do valor, o tipo da variável. Se elas tiverem o mesmo valor, mas tipos diferentes, nos será retornado um **false**.

```
var comparacaoIdentica = '0' === 0
```

### Operadores aritméticos:

- + adição
- - subtração
- \* multiplicação
- / divisão real
- % divisão inteira
- \*\* potenciação

### Operadores relacionais

Consultam a relação entre valores, nos retornando uma afirmação

- > maior que;
- < menor que;
- >= maior ou igual a;
- <= menor ou igual a;

### Operadores lógicos

São tipos de operadores que consultam valores lógicos:

- && - “e” - precisa que todos os valores sejam true
- || - “ou” - considera qualquer valor que seja true
- ! - “não” - inverte o valor de true para false ou vice-versa

## Vetores e objetos

### Vetores ou arrays:

Arrays são um tipo de lista, ou matriz de variáveis, onde cada variável possui um índice.

Os valores podem ser de vários tipos.

É uma caixa com várias outras caixas dentro e cada uma contendo algum valor.

ex:

```
let array=['string',1,true...]
```

Arrays devem ser declarados entre colchetes [] e podem guardar qualquer valor dentro de seus índices, inclusive outros arrays.

ex:

```
let array = ['string',1,true,false,['array1'],['array2']...]
```

para irmos a uma seção específica do array, podemos usar:

```
console.log(array[3]);
```

### Manipulando Arrays

Ao ser declarado, o Array traz consigo uma série de métodos para manipulá-lo.

vamos usar o seguinte array como exemplo:

```
let array = ['string', 1,true, ['array1'], ['array2']]
```

- **forEach()**

Executará uma função para cada índice do array

ex:

```
array.forEach(function(item, index){console.log(item, index)})
```

para cada índice de um array chamado convenientemente de *array*, ele irá nos mostrar no log qual o item e o número do índice.

string	0	script.js:5
1	1	script.js:5
true	2	script.js:5
▶ ['array1']	3	script.js:5
▶ ['array2']	4	script.js:5

- **push()**

Adiciona item no final do array

ex:

```
array.push('novo item')
```

Ao fazermos isso e checarmos o console, nos aparecerá, no final, o 'novo item'

```
▶ (6) ['string', 1, true, Array(1), Array(1), 'novo item']
```

- **pop()**

Remove item no final do array

ex:

```
array.pop()
```

o elemento "array 2" foi apagado após fazermos isso

```
▶ (4) ['string', 1, true, Array(1)]
```

- **shift()**

Remove item no início do array

ex:

```
array.shift()
```

o elemento "string" do nosso array foi removido

```
[1, true, Array(1), Array(1)]
```

- **unshift()**

Adiciona item no início do array

ex:

```
array.unshift('teste');
```

o 'teste' foi inserido no começo do array

```
▶ (6) ['teste', 'string', 1, true, Array(1), Array(1)]
```

- **indexOf()**

Retorna o índice de um valor

ex:

```
console.log(array.indexOf(true))
```

o console nos retornará o índice em que 'true' está

```
2
```

- **splice()**

remove ou substitui uma série de itens

ex:

```
array.splice(0,3)
```

decidimos que o que está entre o índice 0 e antes do índice 3 será removido do nosso array, restando apenas os dois últimos valores.

```
▶ (2) [Array(1), Array(1)]
```

- **slice()**

retorna uma parte de um array existente

ex:

podemos criar outro array baseado no array já existente, supondo que só queremos o índice 0, 1 e 2 dele:

```
let novoArray = array.slice(0,3)
```

isso nos retornaria os 3 primeiros valores.

```
▶ (3) ['string', 1, true]
```

(embora, no slice e splice botemos índice 0 ao 3, eles contarão apenas até o número anterior ao último índice selecionado. Caso quiséssemos que o índice 3 fosse incluído, deveríamos botar 0,4)

- **Spread**

Uma forma de lidar separadamente com os elementos de um array, o que fizer parte de um array vai se tornar um elemento independente. Deve-se utilizar ... na frente da variável. É usado quando estamos chamando uma função.

```
function sum(x, y, z) {  
  return x + y + z;  
}  
  
const numbers = [1, 2, 3];  
  
console.log(sum(...numbers));
```

- **Rest**

Combina os argumentos num array. O que era independente agora se torna um array. É utilizado quando se declara uma função

```

function confereTamanho(...args) {
  console.log(args.length)
}

confereTamanho() // 0
confereTamanho(1, 2) // 2
confereTamanho(3, 4, 5) // 3

```

## Objetos

São dados que possuem propriedades e valores que definem suas características. Deve ser declarado entre chaves “{}”.

Ex.: uma xícara azul, além da cor, pode ter vários tamanhos e funções, podendo ser declarada da seguinte forma:

```

var xicara = {
  cor:'azul',
  tamanho: 'p',
  funcao: tomarCafe()
}

```

## Manipulando objetos

Suas propriedades podem ser atribuídas a variáveis, facilitando a manipulação do objeto.

**Isto é uma desestruturação.**

ex:

```

var xicara = {cor:'azul', tamanho:'p', funcao:tomarCafe()}
var cor=xicara.cor;
var tamanho = xicara.tamanho;
var funcao = tomarCafe();

```

objetos podem conter outros objetos dentro de si. No VSCode, ficará mais ou menos dessa maneira:

```

let object = {texto: 'string', number: 1, boolean: true, array:
['array'], objectInterno:{objectInterno: 'objeto interno'}};

```

para navegarmos pelo objeto, podemos usar o seguinte comando:

```

console.log(object.number)

```

desta forma, o log nos mostrará o que está contido na propriedade ‘number’ de nosso objeto: **1**

para desestruturarmos, seguiremos essa lógica da navegação.

Se quiséssemos criar uma variável que contivesse apenas o que está incluso na propriedade ‘texto’, faríamos o seguinte:

```

texto = object.texto

```

se checarmos no log o que teríamos na variável ‘texto’, veríamos que ela contém o que temos na propriedade ‘texto’ de nosso objeto:

```

string

```

Outra forma de desestruturar é a seguinte:

na hora de declararmos uma variável, podemos abrir chaves que se relacionam ao nosso objeto:

```
var {} = object
```

dentro das chaves, botaríamos o que esta variável receberia de nosso objeto, separado por vírgulas:

```
var {texto, number, boolean} = object
```

para checarmos isso num log, o comando é um pouco diferente:

```
console.log(texto, number, boolean)
```

```
string 1 true
```

## Estruturas condicionais

### O que são estruturas condicionais?

#### Estruturas condicionais

São instruções para realizar tarefas a partir de uma condição, seja de decisão ou de repetição.

ex: Um jogo precisa mudar o placar toda vez que um jogador marca pontuação.

#### Estruturas de decisão

- **If**

if significa “se”. Ele precisa que alguma condição seja atendida para se fazer valer, por exemplo:

considerando as seguintes variáveis:

```
var jogador1 =0;  
var jogador2 =0;  
var placar;
```

num jogo, se um jogador marcar um ponto, poderíamos fazer a seguinte lógica:

```
if (jogador1 > 0){  
    console.log('Jogador 1 marcou  
    ponto')  
}  
  
if (jogador2 > 0){  
    console.log('Jogador 2 marcou  
    ponto')  
}
```

- **Else**

No caso de uma condição não ser atendida, podemos usar o else:

```
if (jogador1 > 0){  
    console.log('Jogador 1 marcou  
    ponto')  
} else {  
    console.log('O jogador 1 ainda não  
    marcou ponto')  
}
```

- **Else if**

Caso haja mais de uma condição relacionada, utilizamos o else if, que é a junção dos dois:

```

if (jogador1 > 0){
    console.log('Jogador 1 marcou
    ponto')
} else if (jogador2 > 0){
    console.log('Jogador 2 marcou
    ponto')
} else {
    console.log('ninguém marcou ponto')
}

```

- **Ninho de if**

Pode-se usar, também, um if dentro de outro:

ex:

Visando verificar a validade do jogo, este trecho só irá rodar quando a pontuação for maior que -1

```

if (jogador1 > -1){
    if (jogador1 > 0){
        console.log('Jogador 1 marcou
        ponto')
    } else if (jogador2 > 0){
        console.log('Jogador 2 marcou
        ponto')
    } else {
        console.log('ninguém marcou
        ponto')
    }
}

```

- **If ternário**

Podemos também fazer uma verificação em uma única linha usando “if” ternário:

ex: [condição]?[instrução1]:[instrução2];

a condição do if ternário sempre acaba com um ? e o else é representado pelo :

ex:

```

jogador1 > -1 && jogador2 > -1? console.log('os jogadores são válidos'):(os jogadores são inválidos');

```

- **Usando switch/case**

ele também funciona como uma estrutura condicional. deve sempre ser finalizado com um break para que a função pare e não se torne um loop infinito. Considera tanto o valor quanto o seu tipo (Ex: um ‘1’ como string ou como number).

```

switch(placar) {
    case placar = jogador1 > jogador2 :
        console.log('jogador 1 ganhou');
        break;
    case placar = jogador2 > jogador1 :
        console.log('jogador 2 ganhou');
        break;
    default:
        console.log('ninguém marcou
        ponto')
}

```

## Laços de repetição

São estruturas condicionais que repetem uma instrução até atingir determinada condição:

- **For**

Funciona como uma repetição de instrução até que a condição seja falsa:

ex: consideremos o array e object abaixo:

```
let array = ['valor1', 'valor2', 'valor3', 'valor4']
let object= {propriedade1: 'valor1', propriedade2: 'valor2', propriedade3: 'valor3'}
```

```
for (let indice = 0; indice < array.length; indice++){
```

Nesse caso, considerando que a variável indice comece em zero, enquanto o indice for menor que o tamanho do array, acrescentaremos +1 a ele.

- **For/in**

Funciona como uma repetição a partir de uma propriedade:

```
for (let i in array) {
  console.log(i);
}
```

usando nossa variável i in array, o log nos imprimirá que o i equivale a uma série de strings iguais aos índices do array:

```
0
1
2
3
```

utilizando o for in no object, ele irá nos retornar, no log, as propriedades do objeto como string: Para pegarmos o valor, substituímos o () pelo [].

```
for (i in object) {
  console.log(i)
}
```

```
propriedade1
propriedade2
propriedade3
Live reload enabled.
```

- **For/of**

Funciona como uma repetição a partir de um valor:

“para um índice de um array...”

No exemplo abaixo, para cada índice do array, mostraremos no log seu conteúdo

```
for (i of array) {
  console.log(i);
}
```

```
valor1
valor2
valor3
valor4
```

O for/of não funciona com objetos pois suas propriedades variam, diferentes do índice em um array que sempre serão números inteiros.

Caso queiramos usar o for of num objeto mesmo assim, deveremos usar em sua **propriedade**, como no seguinte exemplo

```
for (i of object.propriedade2) {
  console.log(i)
}
```



Mas, quando checarmos o log, veremos o problema: ele mostrará cada letra numa linha só:

```
v
a
1
o
r
2
```

- **While**

Executa uma instrução **enquanto** uma determinada condição for verdadeira. Sua verificação é feita antes da execução.

ex: enquanto a variável a for menor que 10 ela irá receber +1

```
var a=0;
while (a<10) {
  a++;
  console.log(a)
}
```

1	script.js:61
2	script.js:61
3	script.js:61
4	script.js:61
5	script.js:61
6	script.js:61
7	script.js:61
8	script.js:61
9	script.js:61
10	script.js:61
Live reload enabled.	
	index.html:40

- **Do/while**

Executa uma instrução **até que** uma determinada condição seja falsa, a verificação é feita depois da execução; O do while sempre será executado ao menos uma vez!

```
var a = 0
do {
  a++;
  console.log(a);
} while (a<10)
```

1	script.js:66
2	script.js:66
3	script.js:66
4	script.js:66
5	script.js:66
6	script.js:66
7	script.js:66
8	script.js:66
9	script.js:66
10	script.js:66

## Funções e suas particularidades

### Funções:

São blocos de comandos e instruções para a execução de determinadas tarefas:

ex: `function nomeDaFuncao(){`  
 `${instrucao};`  
`}`

`nomeDaFuncao()`

```
function funcao(){
  console.log('tudo ok');
}

funcao();
```

tudo ok
Live reload enabled.

Para declarar, usamos a palavra reservada **'function'** seguida do **nome de nossa função** ao lado de **()**, que indica que é um objeto do tipo **function**, e seguido por **{}**, que significa um bloco de instrução.

### Funções com parâmetros:

As funções podem receber em sua declaração parâmetros que servem como variáveis,

onde sua atribuição pode ser feita durante a chamada da função:

ex: `function nomeDaFuncao(parametro){  
 ${instrucao}  
}`

`nomeDaFuncao(valorDoParametro);`

```
function mensagem (primeiro, segundo){  
  console.log(primeiro,segundo)  
}
```

```
mensagem('tudo certo','jovem')
```

tudo certo jovem

## Aprofundando em funções

### Tipos de função:

- **Funções declarativas**

São funções que possuem o uso mais comum, deve ser declarada usando a palavra reservada “function” seguida do **nome da função**, parênteses () e chaves {}

O nome da função é obrigatório:

```
function funcao(){  
  console.log('tudo ok');  
}  
  
funcao();
```

ela deve ter um nome obrigatoriamente

- **Expressões de Funções**

São funções atribuídas às expressões. A nomeação das funções por expressão é opcional:

ex1: `var funcao = function nomeDaFuncao(){  
 ${instrucao};  
}`

ex2: `var funcao = function(){  
 ${instrucao};  
}`

```
var funcao = function funcao () {  
  console.log('sou uma funcao')  
}  
  
funcao()
```

sou uma funcao

caso retiremos o nome “funcao” de function, ainda assim ela irá funcionar

```
var funcao = function() {  
  console.log('sou uma funcao')  
}  
  
funcao()
```

- **Arrow Functions:**

São funções de expressão de sintaxe curta. Arrow functions sempre serão anônimas, portanto, não podem ser nomeadas. Deve ser declarada com parênteses (), seguido de => e depois chaves {}

ex: `var funcao = () => {  
 ${instrucao};  
}`

```

}
var funcao = ()=> {
  console.log('olá!')
}
funcao()

```

olá!

Arrow functions não fazem hoisting! This não funciona aqui. Também não existe o objeto “arguments” e nem o construtor pode ser utilizado.

Caso queiramos inserir variável e expressões às nossas strings, utilizamos as crases `` ao invés de aspas.

```
alert(`${n1}+${n2}=${resultado}`)
```

Utilizamos o ! na frente de uma variável para checarmos se ela difere do tipo esperado:

```

//Número, será inválido
if(!operacao || operacao >= 7){
  alert('Erro- operação inválida');
  calculadora()
} else{

```

Neste caso, a **constante** operacao é dada em números, conforme mostra o exemplo abaixo. dentro do if, a ! serve como um modo de checar se o que botamos dentro desta constante foi um número ou não.

```

const operacao = Number(prompt('Escolha
uma opção:\n 1-Soma (+)\n 2-Subtração (-)
\n 3-Multiplicação(*)\n Divisão Real (/)
\n 5-Divisão Inteira (%)\n 6-Potenciação
(**) '));

```

No mesmo caso superior, podemos observar o emprego do ‘||’. O || é um equivalente ao “ou” de nossa língua.

## Prática: Criação de Calculadora

```

//criar uma função chamada "calculadora". após ela, criar uma
constante chamada de 'operação' que nos mostrará um prompt com o
seguinte texto:
function calculadora() {
  const operacao = Number(prompt('Escolha uma opção:\n 1-Soma (+)\n
2-Subtração (-)\n 3-Multiplicação(*)\n Divisão Real (/)\n 5-Divisão
Inteira (%)\n 6-Potenciação (**) '));
  console.log(operacao);

  //Vamos checar a validade de nossa operação, se a pessoa escolheu de
1 a 6.
  //o ! na frente do 'operacao' vai checar se ele não é um número, se
ele não for um número, será inválido
  if(!operacao || operacao >= 7){
    alert('Erro- operação inválida');
    calculadora()
  } else{

```

```
//nós usamos o Number na frente do prompt para convertermos o
resultado que botarmos aqui em número
//esses são os números e os resultados, que serão calculados e
mostrados para a gente!
let n1 = Number(prompt ('Insira o primeiro valor:'));
let n2= Number(prompt ('Insira o segundo valor'));
let resultado;

if(!n1 || !n2){
    alert('Erro- parâmetros inválidos')
    calculadora()
} else{
//as funções dos cálculos da calculadora ficam aqui
function soma(){
    resultado = n1 + n2;
    alert(`${n1}+${n2}=${resultado}`);
    novaOperacao()
}

function sub(){
    resultado = n1 - n2;
    alert(`${n1}-${n2}=${resultado}`);
    novaOperacao()
}

function mult(){
    resultado = n1 * n2;
    alert(`${n1}*${n2}=${resultado}`);
    novaOperacao()
}

function divReal(){
    resultado = n1 / n2;
    alert(`${n1}/${n2}=${resultado}`);
    novaOperacao()
}

function divInt(){
    resultado = n1 % n2;
    alert(`o resto da divisão entre ${n1}e ${n2} é ${resultado}`);
    novaOperacao()
}
```

```

function pot(){
    resultado = n1 ** n2;
    alert(`${n1} elevado a ${n2} é igual a ${resultado}`);
    novaOperacao()
}

// para evitar dar refresh para fazer outra conta, podemos criar uma
função que resete a calculadora após acabar

function novaOperacao () {
    let opcao = prompt('Deseja fazer outra operação?\n 1- Sim\n
2-Não');

    if(opcao == 1){
        calculadora();
    }else if (opcao == 2){
        console.log('Até mais!')
    } else {
        alert('Digite uma opção válida!')
        novaOperacao();
    }
}

// a partir do número que botarmos no prompt inicial, a calculadora
irá iniciar uma função diferente, voltada para que operação queremos
if (operacao ==1){
    soma();
} else if (operacao ==2) {
    sub();
} else if (operacao ==3) {
    mult()
} else if (operacao ==4) {
    divReal()
} else if (operacao ==5) {
    divInt()
} else if (operacao ==6) {
    pot()
}
}
}

```

```
calculadora()
```

## SINTAXE E OPERADORES

### Operadores

#### Tipos de operadores:

Sinais utilizados para manipular um certo valor ou validar uma certa condição.

Os mais utilizados são:

- **Operadores de atribuição**

Operador	Exemplo	Equivalente a
=	x=y	x=y
+=	x+=y	x=x+y
*=	x*=y	x=x*y
/=	x/=y	x=x/y
%=	x%=y	x=x%y

- **Operadores de aritmética**

Operador	Exemplo
+	Adição
-	Subtração
*	Multiplicação
**	Exponencial
/	Divisão
%	Módulo
++	Incrementar
--	Decrementar

- **Operadores de comparação**

Operador	Descrição
==	Igual a
===	Mesmo valor e mesmo tipo
!=	Diferente
!==	Valor e tipos diferentes

>	Maior que
<	Menor que
>=	Maior ou igual
<=	Menor ou igual

- Operadores de lógica

Operador	Descrição
&&	“e” lógico
	“ou” lógico
!	“não” lógico

- Operadores condicionais

Ternário
(condição) ? expressão1 : expressão2
Equivale a: if (condição){expressão 1;} else {expressão2;}

## VARIÁVEIS E TIPOS

### Atribuindo Valores

#### Tipos de case:

- **String original**  
um exemplo aqui
- **Camel Case**  
umExemploAqui
- **Snake Case**  
um\_exemplo\_aqui
- **Kebab Case**  
um-exemplo-aqui
- **Pascal Case**  
UmExemploAqui
- **Upper case snake case**  
UM\_EXEMPLO\_AQUI (muito usado em constantes)

#### Variáveis:

Para declarar algum valor que será mutável, podemos usar a **var** ou a **let**

O **let** só funciona dentro de um bloco, ex: dentro de uma função

Variáveis são declaradas usando o Camel Case.

#### Hoisting:

É o deslocamento de todas as variáveis e funções para o topo do código. Não funciona com o **let**. Para funcionar com o **let**, ele deve ser declarado no começo.

Com o **let**, devemos nos atentar: o código vai diferenciar variáveis de escopo global ou

escopo de bloco.

### Constantes:

As constantes são declaradas em SNAKE\_UPPER\_CASE, possuindo escopo de bloco e não fazendo hoisting. O valor da constante não muda! Sem poder ser redeclarado ou reatribuído. Por não fazer hoisting, nós devemos manualmente posicionar as constantes no começo do código.

	var	const	let
escopo	global ou local	bloco	bloco
redeclarar	sim	não	não
reatribuir	sim	não	sim
hoisting	sim	não	não

exemplo de reatribuição: se temos uma variável chamada **var v1 = 10** e de queremos mudar seu valor: **v1 = 15**

exemplo de redeclaração: se temos uma variável chamada **var v1 = 10** e a declaramos novamente: **var v1 = 15**

## Tipos

### Estruturas de dados:

Fazem parte de uma linguagem de tipagem dinâmica. Antes de declarar um valor, não precisamos declarar o seu tipo.

Podemos usar o comando `typeof` **variável** para saber seu tipo

### Tipos de dados no javascript:

Dividem-se em **tipos primitivos** e **tipos não-primitivos ou compostos**

- **Tipos primitivos**

- **Números**

- Números inteiros ou decimais
  - Interage com operações aritméticas
  - Temos a biblioteca chamada Math, que possui uma série de valores matemáticos
  - Arredondar para cima: **Math.ceil(variavel)**
  - Arredondar para baixo: **Math.floor(variavel)**
  - Para mostrarmos uma porcentagem, devemos trabalhar com a concatenação de números e strings, pois o % representa o resto de uma divisão
  - Para transformarmos um número numa string, usamos: **variavel.toString()**

- **Strings**

- Comumente utilizadas para texto
  - Valores declarados entre aspas ou crases
  - Podemos juntar duas strings: ``Nome completo: ${nome1} ${nome2}``
  - **Length**: Podemos ver o índice e o tamanho dos strings com os comandos: **variavel[numeroDoIndice]** e **variavel.length**
  - **Concatenação**: Para concatenar dois strings ou mais, usamos: **variavel1.concat(variavel2)** no entanto, os nomes juntos não terão espaços
  - **Formatação**: Para espaçarmos, podemos atribuir uma nova variável e nela somar strings: **espacado = variavel1 + " " + variavel2**. no entanto, o jeito mais fácil seria: **espacado = `\${variavel1} \${variavel2}`**
  - Para separarmos todos os elementos de uma frase, podemos utilizar o **split**, ex: **variavel.split(" ")**. Caso inseríssemos espaço dentro destas



- aspas(" "), ele separaria a frase pelos espaços
- Para checarmos se a frase inclui um elemento, usamos o comando: **variavel.includes("elemento")**
- Para checarmos se a frase começa com um elemento, usamos o comando: **variavel.startsWith("letra")**
- Para checarmos se a frase termina com um elemento, usamos o comando: **variavel.endsWith("letra")**

#### Booleans

- Utilizados para checar se um valor é verdadeiro ou falso

#### • Tipos não-primitivos

##### Objetos

- São declarados na seguinte formatação: **let nomeDoObjeto = {chave1:valor1, chave2:valor2};**
- As chaves são estas propriedades do objeto
- Para adicionarmos uma chave ao objeto, fazemos: **nomeDoObjeto.chave=valor** ou **nomeDoObjeto["chave"]= valor**
- Podemos checar a biblioteca de Object do Javascript, que contém uma série de interações com o objeto, ex: **Object.values(nomeDoObjeto)**
- Se quisermos consultar uma chave de um objeto, podemos usar: **nomeDoObjeto.chave**

##### Arrays

- São listas iteráveis de elementos, conhecidas também como vetores
- Para adicionarmos um valor ao final do array, usamos: **nomedoarray.push(valor)**
- Para retirarmos um valor do final do array, usamos: **nomedoarray.pop(valor)**
- Para retirarmos um valor do começo do array, usamos: **nomedoarray.shift(valor)**
- Para acrescentarmos um valor ao começo do array, usamos: **nomedoarray.unshift(valor)**
- Para vermos se o array inclui algo, usamos **nomedoarray.includes(valor)**
- Para ver se todos os itens são o mesmo valor: **nomedoarray.every(valor)**
- Para checar apenas se algum dos itens têm determinado valor: **nomedoarray.some(valor)**
- Para inverter um array: **nomedoarray.reverse()**

#### • Outros tipos

##### Null

- O objeto com "null" tem um valor nulo

##### Undefined

- Não há nada atribuído nem definido em algo undefined

##### Empty

- São valores "vazios": uma variável 0, uma string sem nada nas ""

## FUNÇÕES

### Estrutura e função anônima

#### Estrutura de uma função:

```
function nome(parametros) {
  // instruções
}
```

As variáveis criadas dentro de uma função somente poderão ser utilizadas dentro dela.

O comando **return** para de executar uma função e retorna o valor que vem após ele.

### Função anônima:

são funções que representam expressões:

```
const soma = function (a, b) {  
  return a + b;  
};  
ex:
```

## Função autoinvocável e callbacks

### Função autoinvocável:

É uma função anônima entre parênteses, seguida por outro par de parênteses que representará sua chamada.

```
(  
  function() {  
    let name = "Digital Innovation One"  
    return name;  
  }  
)();  
  
const soma3 = (  
  function() {  
    return a + b;  
  }  
) (1, 2);  
  
console.log(soma3) // 3
```

Também pode ser utilizada com parâmetros.

### Callbacks:

São funções utilizadas como parâmetro para outras funções. Usando callback, temos um maior controle da ordem de chamadas

```
const calc = function(operacao, num1, num2){  
  return operacao(num1, num2);  
}  
  
const soma = function(num1, num2) {  
  return num1 + num2;  
}  
  
const sub = function(num1, num2) {  
  return num1 - num2;  
}  
  
const resultSoma = calc(soma, 1, 2);  
const resultSub = calc(sub, 1, 2);  
  
console.log(resultSub); // -1  
console.log(resultSoma); // 3
```

## Parâmetros e Loops

Os conceitos desta aula foram para o capítulo **SINTAXE BÁSICA EM JAVASCRIPT**, uma vez que são idênticos e não faria sentido repeti-los

## This

### O que é This?

É uma referência de contexto.

```
const pessoa = {  
  firstName: "André",  
  lastName: "Soares",  
  id: 1,  
  fullName: function() {  
    return this.firstName + " " + this.lastName;  
  },  
  getId: function() {  
    return this.id;  
  }  
};
```

```
pessoa.fullName();  
// "André Soares"  
  
pessoa.getId();  
// 1
```

No exemplo acima, **this** é utilizado para se referir ao objeto **pessoa**

O seu valor pode mudar de acordo com o lugar no código onde foi chamada.

Contexto	Referência
Em um objeto (método)	Próprio objeto dono do método
Sozinha	Objetos globais (em navegadores, o window)
Função	Objeto global
Evento	Elemento que recebeu o evento

## Manipular o valor do This?

### Call

Podemos criar uma função, e em seus parâmetros inserir um this relacionado a alguma propriedade ou variável. Ao chamarmos esta função, especificamos a o que esta função irá se relacionar:

```
const pessoa = {
  nome: 'Miguel',
};

const animal = {
  nome: 'Murphy',
};

function getSomething() {
  console.log(this.nome);
}

getSomething.call(pessoa);
```

Também podemos passar parâmetros para a call.

```
const myObj = {
  num1: 2,
  num2: 4,
};

function soma(a, b) {
  console.log(this.num1 + this.num2 + a + b);
}

soma.call(myObj, 1, 5);
// 12
```

### Apply

É semelhante ao call, mas, ao referenciar argumentos, eles serão passados dentro de um array.

```
const myObj = {
  num1: 2,
  num2: 4,
};

function soma(a, b) {
  console.log(this.num1 + this.num2 + a + b);
}

soma.apply(myObj, [1, 5]);
// 12
```

## Bind

Clona a estrutura da função onde é chamada e aplica o valor do objeto passado como parâmetro.

```
const retornaNomes = function () {  
  return this.nome;  
};  
  
let bruno = retornaNomes.bind({ nome: 'Bruno' });  
  
bruno();  
// Bruno
```

## DEBUGGING E ERROR HANDLING

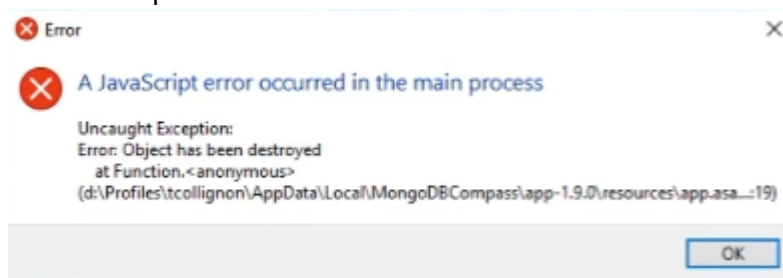
### Tipos de erros

#### ECMAScript Error e DOMException:

##### ECMAScript Error:

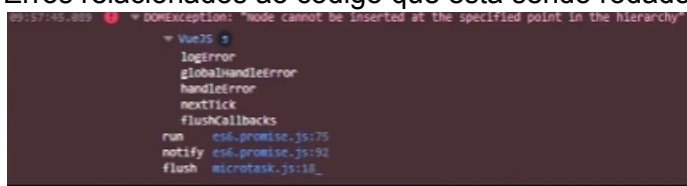
São erros que ocorrem em tempo de execução, composto por:

- Mensagem
- Nome do erro
- Linha do ocorrido
- Call Stack: pilha de chamadas



##### DOMException:

Erros relacionados ao código que está sendo rodado por uma página da web



### Tipos de erros

#### Throw, Try/Catch e Finally:

##### Throw

O throw irá retornar alguma condição que determinamos como um erro, e não como um elemento do código (como uma string).

##### Try/Catch

Dentro do Try, o programa irá verificar o código e se houver um erro ele será pego no catch, manipulando-o da maneira que preferirmos.

```
function verificaPalindromo(string) {
  if (!string) throw "String inválida";

  return string === string.split('').reverse().join('');
}

function tryCatchExemplo(string) {
  try {
    verificaPalindromo(string)
  }
  catch(e) {
    throw e;
  }
}

tryCatchExemplo('');
```

### Finally

Uma instrução que será chamada independente de ter um erro ou não.

## Criando Erros

### Objeto error:

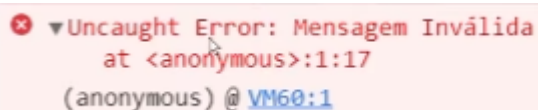
Podemos manipular o que será mostrado num erro mudando o objeto Error:

```
new Error(message, fileName, lineNumber)

// todos os parâmetros são opcionais

const MeuErro = new Error('Mensagem Inválida');

throw MeuErro;
```



Uncaught Error: Mensagem Inválida  
at <anonymous>:1:17  
(anonymous) @ VM60:1

Podemos checar informações como:

- **Nome do erro**  
*nomeDoErro.name*
- **Stack do erro**  
*nomeDoErro.stack*
- **O que é apresentado no erro**  
*nomeDoErro*

## JAVASCRIPT ASSÍNCRONO

### Assincronicidade

#### Definição, Promises e Async/Await:

##### Assincronicidade:

Algo que não ocorre ou não se efetiva ao mesmo tempo.

Por padrão o javascript roda de maneira síncrona. No modo assíncrono, fazemos uma coisa enquanto fazemos outra.

##### Promises:

Objeto de processamento assíncrono. No início, seu valor é desconhecido. Ela pode, então, ser resolvida ou rejeitada.

Uma promise tem três estados:

- **Pendente (pending)**

- Completada (fulfilled)
- Rejeitada(rejected)

```
const myPromise = new Promise((resolve, reject) => {
  window.setTimeout(() => {
    resolve(console.log('Resolvida!'));
  }, 2000);
});
```

## Async/Await

Funções assíncronas precisam dessas duas palavras-chave

```
async function resolvePromise() {
  const myPromise = new Promise((resolve, reject) => {
    window.setTimeout(() => {
      resolve('Resolvida');
    }, 3000);
  });

  const resolved = await myPromise
    .then((result) => result + ' passando pelo then')
    .then((result) => result + ' e agora acabou!')
    .catch((err) => console.log(err.message));

  return resolved;
}
```

Funções assíncronas também retornam promises! E podemos utilizar o try/catch no async/await.

## Consumindo APIS

### O que são APIS e Fetch:

#### O que são APIS?

Significam Application Programming Interface, e são uma forma de intermediar os resultados do back-end com o que é apresentado no front-end. Podem ser acessadas através de URLs.



## JSON

Significa 'JavaScript Object Notation', e é muito comum que APIS retornem seus dados no formato .json, sendo necessário tratarmos estes dados quando recebermos.

## Fetch

O fetch retorna uma Promise, utilizado para tratar uma API

```
fetch(url, options)
  .then(response => response.json())
  .then(json => console.log(json))

// retorna uma Promise
```

## ORIENTAÇÃO A OBJETOS

### Paradigmas e Pilares

#### Paradigmas e Pilares:

Temos dois paradigmas na programação, o **Imperativo** e o **Declarativo**:

O imperativo foca em como você irá resolver os problemas, e o declarativo no que você irá resolver.

Na orientação a objetos, os programas são “objetos” que possuem uma série de propriedades, dentre estas, quatro pilares são essenciais:

- **Herança**  
Relações onde objetos-filho irão herdar propriedades e métodos do objeto-pai
- **Polimorfismo**  
Objetos podem herdar a classe do pai, mas se comportarem de forma diferente quando invocamos seus métodos
- **Encapsulamento**  
Cada classe tem propriedades e métodos independentes do restante do código
- **Abstração**  
O ato de simplificar cada vez mais um objeto complexo

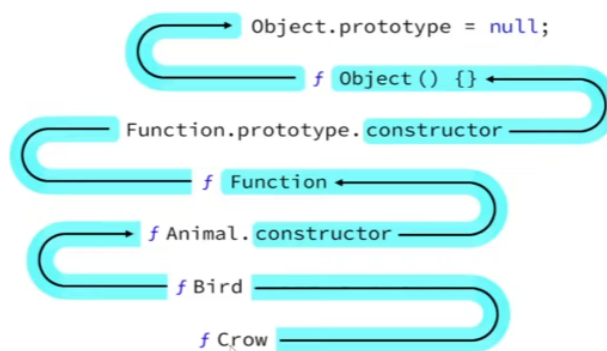
### Orientação a Objetos em JavaScript (OOJS)

#### Protótipos e Classes:

##### Protótipos

Todos os objetos Javascript herdam propriedades e métodos de um prototype. O objeto `Object.prototype` está no topo desta cadeia.

Cadeia de protótipos (prototype chain)



##### Classes

Não existem nativamente no JavaScript. São uma **syntactic sugar**, ou seja, uma sintaxe feita para facilitar a escrita. Todas as classes são objetos e a herança se dá por protótipos.

## MANIPULANDO A D.O.M. COM JAVASCRIPT

### Entendendo o D.O.M.

#### O que é o D.O.M. e D.O.M. vs B.O.M.

##### O que é o D.O.M.?

Significa Document Object Model. o DOM HTML é um padrão de como acessar e modificar elementos html de uma página. São os nós que representam cada elemento html de uma página.

##### D.O.M. vs B.O.M.?

O BOM significa Browser Object Model, representando tudo que está dentro do objeto window. O window faz parte do BOM, mas não do DOM. O documento está contido no B.O.M.

### Selecionando Objetos

#### Métodos:

##### Estrutura de um html

- **tags**  
Tags definem o que a parte do código representa
- **ids**  
Ids são um identificador único, um id só pode ser usado para um elemento.
- **classes**  
Diferente dos ids, as classes são identificadores, mas não são únicos. Podem ser atribuídas a diversos elementos.

Podemos usar uma série de comandos para selecionarmos documentos html, referenciando-os por suas tags, ids ou classes:

1. **document.getElementById('nomeDald')**
2. **document.getElementsByTagName('nomeDaTag')**  
Ele retornará um array com todos os elementos. Para retornar um específico, teríamos que referenciar com colchetes e indicar o índice dele
3. **document.getElementsByClassName('nomeDaClasse')**  
Assim como na alternativa anterior, ele também retornará um array
4. **document.querySelectorAll('.nomesDasClasses1 .nomeDasClasses2')**  
Aqui, os . são os que representam a classe. caso queiramos referenciar uma id, usamos #  
Também podemos referenciar um recorte mais específico, como uma tag que possui uma classe: **document.querySelectorAll("tag .nomeDasClasses1")**
5. **Adicionar e deletar elementos:**

Método	Descrição
<code>document.createElement(element)</code>	Cria um novo elemento html
<code>document.removeChild(element)</code>	Remove um elemento
<code>document.appendChild(element)</code>	Adiciona um elemento
<code>document.replaceChild(new, old)</code>	Substitui um elemento

### Trabalhando com Estilos

#### Trabalhando com Estilos:

##### Classes



Podemos manipular as classes de um elemento através do **Elemento.classList**

**1. Adicionar classe**

`Elemento.classList.add("nome-da-classe")`

**2. Remover classe**

`Elemento.classList.remove("nome-da-classe")`

**3. Ativar/desativar**

`Elemento.classList.toggle("nome-da-classe")`

Ele adicionará esta classe, caso ela não exista, e a removerá caso exista.

Para manipularmos o estilo, é um processo semelhante ao anterior, ex:

- `document.getElementsByTagName('nomeDaTag').style.color="blue"`.  
Estaremos acrescentando o `.style` para especificar que estamos editando um estilo, e após ele especificamos o que exatamente vamos mudar, podendo ser cor, borda e etc.

## Eventos

### Tipos e acionando eventos:

Eventos são qualquer tipo de ação que o usuário faz na página da web.

#### Tipos

- **Eventos de mouse**  
mouseover e mouseout
- **Eventos de clique**  
click e dbclick
- **Eventos de atualização**  
change, load

#### Acionando eventos

Adicionamos ao código uma função chamada Event Listener, criando um evento que será realizado quando o usuário fizer determinada ação.

```
const botao = document.getElementById("meuBotao");  
botao.addEventListener("click", outraFuncao);
```

Podemos colocar diretamente no html:

```
<html>  
<body>  
  
<h1 onclick="mudaTexto(this)"Clique aqui!</h1>  
  
<script>  
  function mudaTexto(id) {  
    id.innerHTML = "Mudei!";  
  }  
</script>  
  
</body>  
</html>
```

diretamente no html, os eventos possuem um "on" no começo. Ex: Ao invés de click, aqui seria onclick.

## INTRODUÇÃO AO REACT NATIVE

### Conhecendo o React Native

#### O que é o React Native?

É um framework open-source do facebook que provém diversos componentes e

complementos. No caso do React Native, traz componentes que são adaptáveis tanto para Android, quanto iOS quanto outras plataformas. Possibilita um desenvolvimento rápido, seguro (devido ao tempo de mercado) e possui diversas vagas!

## Conhecendo a Documentação

### Componentes

#### Componentes básicos

- **Text**  
Um componente para exibir texto.
- **Image**  
Um componente para exibir imagens.
- **TextInput**  
Um componente para se inserir textos via teclado.
- **ScrollView**  
Fornece um contêiner de rolagem que pode hospedar vários componentes e exibições.
- **StyleSheet**  
Fornece uma layer de abstração semelhante às stylesheets do CSS.

#### User interface

- **Button**  
Um componente de botão básico para lidar com toques que devem renderizar bem em qualquer plataforma.
- **Switch**  
Renderiza um input booleano (true/false)

#### List views

- **FlatList**  
Um componente para renderizar listas roláveis de alto desempenho.
- **SectionList**  
Semelhantes à FlatList, mas para listas seccionáveis

#### Componentes e APIs específicos do Android

- **BackHandler**  
Detectar pressões de botão de hardware para navegação back.
- **DrawerLayoutAndroid**  
Renderiza DrawerLayout para Android
- **PermissionsAndroid**  
Fornece acesso ao modelo de permissões introduzido no Android M.
- **ToastAndroid**  
Cria um alerta de Android Toast.

#### Componentes e APIs específicos do iOS

- **ActionSheetIOS**  
API para exibir uma planilha de ações do iOS ou planilha de compartilhamento.

#### Outros

- **ActivityIndicator**  
Mostra um indicador circular de carregamento
- **Alert**  
Inicia uma caixa de diálogo de alerta com o título e a mensagem especificados.
- **Animated**  
Uma biblioteca para criar animações fluidas e poderosas que são fáceis de construir e manter.
- **Dimensions**  
Fornece uma interface para obter as dimensões do dispositivo.

- **KeyboardAvoidingView**  
Fornece uma visualização que sai do caminho do teclado virtual automaticamente.
- **Linking**  
Fornece uma interface geral para interagir com links de aplicativos de entrada e saída.
- **Modal**  
Fornece uma maneira simples de apresentar conteúdo acima de uma exibição de fechamento.
- **PixelRatio**  
Fornece acesso à densidade de pixels do dispositivo.
- **RefreshControl**  
Este componente é usado dentro de um ScrollView para adicionar a funcionalidade pull para atualizar.
- **StatusBar**  
Componente para controlar a barra de status do aplicativo.

Além desses componentes básicos, temos uma série de outros elementos valiosos que podemos ver na documentação: <https://reactnative.dev/docs/components-and-apis>

### APIs

Vemos uma série de APIs já disponíveis neste framework.

Como exemplo, APIs de acessibilidade, animação, dimensões, plataforma, etc.

## O PROCESSO DE INSTALAÇÃO DO REACT NATIVE EXIGE UMA SÉRIE DE PASSOS QUE SÃO MELHOR EXPLICITADOS ATRAVÉS DOS VÍDEOS.

### Conhecendo a Documentação

#### Conhecendo View e Text/Style Sheet

Antes de mais nada, iniciamos criando uma pasta 'src' no diretório de onde o react native foi instalado. Nele, criamos o arquivo javascript de onde ficará nosso app (ex: App.js) e o iniciamos com os seguintes comandos:

```
import React from "react";
import {View, Text, StyleSheet} from "react-native";
```

Nesta segunda linha, escolhemos o que vamos importar do react native. Neste caso, escolhi o View, Text e StyleSheet

Logo após, cria-se o bloco para o App propriamente dito, como uma função

```
const App = () =>{
}
```

Dentro dele, nós retornamos o que gostaríamos de ver. Podemos estilizar o conteúdo importando uma StyleSheet e fazendo uma constante que crie uma stylesheet em nosso projeto, como no exemplo abaixo:

```
//criação de estilo para algum lugar
//separamos os elementos através do ,
const style = StyleSheet.create({
  container:{
    backgroundColor: 'cyan',
    flex:1 // o 1 expande para a tela inteira
  },
  text:{
    fontSize: 50,
```

```

    fontWeight: "bold",
    color: "black"
  }
})

```

Com as opções de estilização definidas, nós adicionamos as propriedades que definimos aos elementos do nosso app:

```

const App = () =>{
  return (
    <View style={style.container}>
      <Text style={style.text}>Hello a todos</Text>
    </View>
  );
}

```

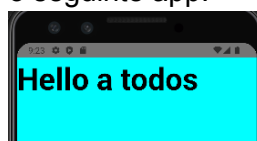
Usando o que aprendemos até então, podemos montar o seguinte código:

```

projects > DIO > dioRn > src > JS App.js > [⌘] style
1  import React from "react";
2  import{View,Text,StyleSheet} from "react-native";
3
4  const App = () =>{
5    return (
6      <View style={style.container}>
7        <Text style={style.text}>Hello a todos</Text>
8      </View>
9    );
10 }
11
12 export default App
13
14 //criação de estilo para algum lugar
15 //separamos os elementos através do ,
16 const style = StyleSheet.create({
17   container:{
18     backgroundColor: 'cyan',
19     flex:1 // o 1 expande para a tela inteira
20   },
21   text:{
22     fontSize: 50,
23     fontWeight: "bold",
24     color: "black"
25   }
26 })

```

Esta combinação nos permite criar o seguinte app:



## Conhecendo SafeAreaView

O SafeAreaView protege o componente de sair para uma área ao qual ele não tem acesso. Para inserirmos, nós adicionamos ele na lista do import

```
import React from "react";
import {View, Text, StyleSheet, SafeAreaView} from "react-native";
```

e, após importarmos, compreendemos o conteúdo do nosso app com o `<SafeAreaView></SafeAreaView>` e transferimos o style da view de container para ele. Também podemos mudar a cor da **status bar** (aquela barrinha que fica na parte superior do celular), importando uma StatusBar e inserindo-a no código com um `<StatusBar/>`.

```
import React from "react";
import {View, Text, StyleSheet, SafeAreaView, StatusBar} from "react-native";
```

Dentro da própria `<StatusBar/>`, escolhemos como estilizá-la.

Ex: No conteúdo anterior, faríamos isso:

```
const App = () =>{
  return (
    <SafeAreaView style={style.container}>
      <StatusBar backgroundColor={'pink'} barStyle='dark-content' />
      <View>
        <Text style={style.text}>Hello a todos</Text>
      </View>
    </SafeAreaView>
  );
}
```

## Conhecendo Image

Se quisermos inserir uma imagem em nosso app, podemos importar o Image para nosso código:

```
import React from "react";
import {View, Text, StyleSheet, SafeAreaView, StatusBar, Image} from
"react-native";
```

Após inserirmos, podemos adicionar a tag `<Image/>` autofechada e dentro dela inserir a nossa fonte. Também podemos inserir uma propriedade de style nela que possa regular certas questões como borda, largura da borda, etc.

Para encurtar nossa tag, podemos criar uma constante com o nosso endereço do link e referenciá-la.

```
const colorGitHub='#010409'
const ImageProfGitHub='https://hermes.digitalinnovation.one/users/student/4157fdc5-d657-4293-a052-b5df798f5a82.png'
const App = () =>{
  return (
    <SafeAreaView style={style.container}>
      <StatusBar backgroundColor={colorGitHub} barStyle='light-content' />
      <View>
        <Image accessibilityLabel="Alexandre com uma arara no ombro"
          style={style.avatar}
          source={{uri:ImageProfGitHub}} />
      </View>
    </SafeAreaView>
  );
}
```

Dessa forma, encurtamos espaço no nosso bloco de código.

Para facilitar a possível mudança de cor no background color, criamos uma constante que contenha qual será o valor a ser utilizado. Desta forma, se mudarmos num lugar, mudamos no resto todo.

À imagem que criamos em nossa view, acrescentamos uma etiqueta de acessibilidade que descreverá a pessoas com deficiência auditiva o conteúdo da imagem (`accessibilityLabel`) e opções de estilo (`style={style.avatar}`). Tais opções de estilo foram as seguintes:

```
const style = StyleSheet.create({
  container:{
    backgroundColor: colorGitHub,
    flex:1 // o 1 expande para a tela inteira
  },
  avatar:{
    height:200,
    width: 200,
    borderRadius: 200/2,
  }
})
```

### Flexbox

Anteriormente, criamos no style um espaço para o container, que irá conter todo nosso conteúdo. Agora já podemos passar para o content, que é o conteúdo propriamente dito. No exemplo abaixo, um content foi criado na stylesheet, e ele definirá o alinhamento de nosso conteúdo.

```
content:{
  // o content se refere ao conteúdo da página propriamente
  dito
  alignItems:"center",
},
```

No bloco de código do app, ficou desta maneira:

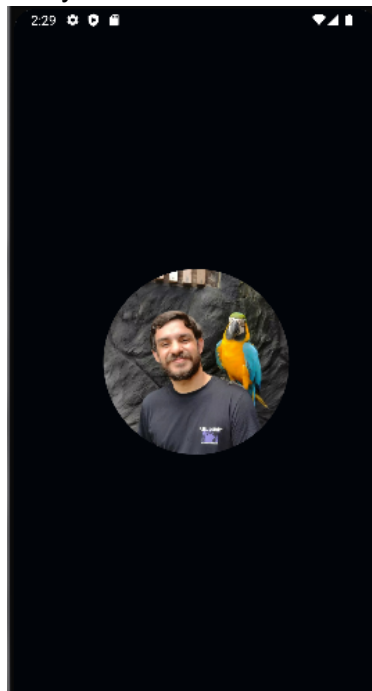
```
const App = () =>{
  return (
    <SafeAreaView style={style.container}>
      <StatusBar backgroundColor={colorGitHub} barStyle='light-content' />
      <View style={style.content}>
        <Image accessibilityLabel="Alexandre com uma arara no ombro"
          style={style.avatar}
          source={{uri:ImageProfGitHub}} />
      </View>
    </SafeAreaView>
  );
}
```

Para alinharmos o nosso conteúdo no centro do container, devemos mudar no próprio container.

Além disso, nossas views se organizam no container em coluna, ou seja, uma embaixo da outra. Caso queiramos que elas se organizem horizontalmente, numa linha, devemos usar o comando **`flexDirection:'row'`**.

```
const style = StyleSheet.create({
  container:{
    //as views se organizam como colunas, se alinhando abaixo da outra. Caso quiséssemos ajustar isso, deveremos
    inserir no container um flexDirection:'row'
    backgroundColor: colorGitHub,
    flex:1, // o 1 expande para a tela inteira
    justifyContent: 'center', // enquanto no content ele alinhou horizontalmente, aqui ele alinha verticalmente
  },
  content:{
    // o content se refere ao conteúdo da página propriamente dito
    alignItems:"center",
  },
  avatar:{
    height:200,
    width: 200,
    borderRadius: 200/2,
  }
})
```

Com todas essas opções, nosso stylesheet fica desta forma, e nosso app assim:



Podemos organizar os flexbox em linha, coluna e definir as proporções que eles irão ocupar a tela. Podemos justificá-los e alinhá-los conforme nosso gosto.

### Adicionando nome, nickname e descrição

Para fazermos o nome, nickname e descrição, devemos adicionar texto em nosso bloco do aplicativo. para tal, adicionamos `<Text></Text>`, e inserimos o texto que queremos no meio. Para separarmos os textos, na linha abaixo, criamos outro `<Text></Text>`.

Ex:

```
const App = () =>{
  return (
    <SafeAreaView style={style.container}>
      <StatusBar backgroundColor={colorGitHub}
barStyle='light-content'/>
      <View style={style.content}>
        <Image accessibilityLabel="Alexandre com uma arara no
ombro"
          style={style.avatar}/>
      </View>
    </SafeAreaView>
  )
}
```

```

        source={{uri:ImageProfGitHub}}/>
        <Text >Alexandre Filho</Text>
        <Text>@pessouza</Text>
        <Text>Estudante de ADS</Text>
    </View>
</SafeAreaView>
);
}

```

Dentro do texto, podemos adicionar estilos diferentes, que podem se juntar com outros complementares num array

```

textoPadrao:{
    color:colorFont
},
//o nome será a fonte de mais destaque, ele irá pegar as características do texto padrao e acrescentar essas
nome:{
    fontWeight:'bold',
    fontSize:25,
},
//nickname será para o apelido, ele irá pegar as características do texto padrao e acrescentar essas
nickname:{
    fontSize:20,
    color: colorFontDark
},
//para nossa descrição, ele irá pegar as características do texto padrao e acrescentar essas
descricao:{
    fontWeight:'bold',
},

```

Na stylesheet, adicionamos um estilo para texto padrão, e outros específicos que o complementam dependendo da situação. No app, inserimos o seguinte:

```

const App = () =>{
    return (
        <SafeAreaView style={style.container}>
            <StatusBar backgroundColor={colorGitHub} barStyle='light-content' />
            <View style={style.content}>
                <Image accessibilityLabel="Alexandre com uma arara no ombro"
                    style={style.avatar}
                    source={{uri:ImageProfGitHub}}/>
                <Text style={[style.textoPadrao, style.nome]}>Alexandre Filho</Text>
                <Text style={[style.textoPadrao, style.nickname]}>@pessouza</Text>
                <Text style={[style.textoPadrao, style.descricao]}>Estudante de Análise e Desenvolvimento de Sistemas</Text>
            </View>
        </SafeAreaView>
    );
}

```

### Criando botão

Para fazer um botão, importamos a tag **Pressable**, e adicionamos ao nosso bloco de código do aplicativo. Dentro dela, montamos uma view que pode incluir o que queiramos que seja pressionável.

No exemplo abaixo, foi um texto:

```

<Pressable onPress={() =>console.log('github')}>
    <View style={style.content}>
        <Text style={style.botao}>Abrir no GitHub</Text>
    </View>
</Pressable>

```



Como exemplificado acima, precisamos adicionar uma função onPress para que ele funcione, caso contrário, nada acontecerá. Nesse caso, ao pressionarmos, ele irá fazer um console log do texto 'github'.

Estilizamos o nosso botão da seguinte maneira:

```
botao:{
  alignItems:'center',
  backgroundColor:'white',
  color:colorGitHub,
  borderRadius:10,
  padding:20,
  fontWeight:'bold',
}
```

O bloco de código do aplicativo ficou da seguinte maneira:

```
const App = () =>{
  return (
    <SafeAreaView style={style.container}>
      <StatusBar backgroundColor={colorGitHub} barStyle='light-content'/>
      <View style={style.content}>
        <Image accessibilityLabel="Alexandre com uma arara no ombro"
          style={style.avatar}
          source={{uri:ImageProfGitHub}}/>
        <Text accessibilityLabel="Nome: Alexandre Filho"
          style={[style.textoPadrao, style.nome]}>Alexandre Filho</Text>
        <Text accessibilityLabel="Nickname do GitHub: pessouza"
          style={[style.textoPadrao, style.nickname]}>@pessouza</Text>
        <Text accessibilityLabel="Alexandre é estudante de Análise e Desenvolvimento de Sistemas"
          style={[style.textoPadrao, style.descricao]}>Estudante de Análise e Desenvolvimento de Sistemas</Text>
      </View>
      <Pressable onPress={()=>console.log('github')}>
        <View style={style.content}>
          <Text style={style.botao}>Abrir no GitHub</Text>
        </View>
      </Pressable>
    </SafeAreaView>
  );
}
```

### Linking e código nativo android

Nesta seção, a aula foi bastante complexa. Irei citar aqui que foi adicionado, mas de forma mais sintética:

1. Uma constante com nosso link do github

```
//url do github
const githubUrl='https://github.com/pessouza';
```

2. Uma função assíncrona dentro de nosso bloco de código do app

```
const handlePressGoToGithub= async () => {
  console.log('verificando link')
  const res= await Linking.canOpenURL(githubUrl);
  if(res){
    console.log('link aprovado')
    console.log('abrindo link...')
    await Linking.openURL(githubUrl)
  }
}
```

3. Relacionamos essa função no nosso onPress

```

<Pressable onPress={handlePressGoToGithub}>
  <View style={style.content}>
    <Text style={style.botao}>Abrir no GitHub</Text>
  </View>
</Pressable>

```

4. Tivemos que ir no arquivo que está o manifest do android, em: android>app>src>main>AndroidManifest.xml, e adicionar o seguinte, acima de <application>:

```

<queries>
  <intent>
    <action android:name="android.intent.action.VIEW"/>
    <category android:name="android.intent.category.BROWSABLE" />
    <data android:scheme="https" />
  </intent>
</queries>

```

5. Após isso, como mexemos no nativo do android, teríamos que reinstalar nosso app, e fazemos isso com a ajuda do yarn. Abrimos o painel de comando, vamos para a pasta de nosso projeto e lá executamos o comando 'yarn android', ex:

```
C:\Users\Alexandre\projects\DIO\dioRn>yarn android
```

6. Após aguardar um pouco, o app será reinstalado com sucesso!

## CRIANDO APLICAÇÕES MÓVEIS MULTIPLATAFORMA NO REACT NATIVE

### Componentes e Estados

#### O que são componentes e propriedades

##### Componentes

Componentes permitem você dividir a UI em partes independentes, reutilizáveis, ou seja, trata cada parte da aplicação como um bloco isolado, livre de outras dependências externas.

##### Propriedades

Os componentes são como funções do JavaScript. Aceitam entradas arbitrárias (chamadas 'props') e retornam elementos que descrevem o que aparecer na tela. exemplo de uso:

```

const Usuario = (props) {
  return <Text>Olá, {props.name}</Text>;
}

```

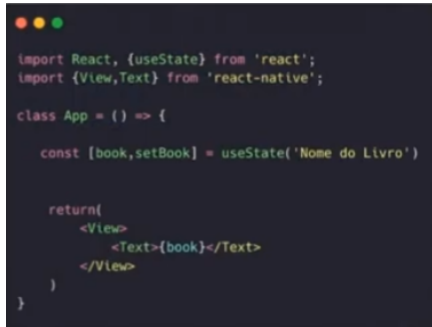
```

const App = () => {
  return <Usuario name="Pablo" />;
}

```

## Estado

Diferente das propriedades, o estado não é repassado ao componente e sim configurado dentro dele. Pense no estado como as propriedades de nossa classe que devem ser armazenadas para renderizarmos o componente da forma correta.



```
import React, {useState} from 'react';
import {View,Text} from 'react-native';

class App = () => {

  const [book,setBook] = useState('Nome do Livro')

  return(
    <View>
      <Text>{book}</Text>
    </View>
  )
}
```

Criamos uma pasta na src com o nome de components, fazendo uma nova pasta dentro dela com o nome que quisermos, seguido por um index.js e um style.js. No index, preenchemos semelhante aos App.js que já fizemos antes, importando o que for preciso e criando uma função com o que queremos mostrar.

No style, fazemos um stylesheet e definimos as opções de estilo.

Para importar o componente a um arquivo, usamos **import Componente from 'localizaçãoDaPasta'** em nosso arquivo principal. chamamos ele, no nosso bloco de código do app, da seguinte maneira: **<Componente;>**

Podemos copiar e colar quantos quisermos, mas para diferenciarmos um do outro, precisaríamos criar propriedades e editá-las.

Dentro no index.js de nosso componente, na definição da constante que será a sua função, entre os parênteses usamos **{propriedades}**. Por exemplo, se criarmos uma propriedade na função card chamada Card (**{titulo}**) e inserirmos no bloco de código desse componente: **<Text> {titulo} </Text>**, podemos decidir no nosso código principal que o o título do componente será **Experiência profissional** se fizermos **<Componente titulo= 'experiencias profissionais'>**.