**Curso de JavaScript - Cursoemvideo**

***FOQUE !!! EXERCITE !!! ANOTE !!!***

**Cursoemvideo**

**Professor: Gustavo Guanabara**

**40h**

**Inicio: 08/01/2024**

**Introdução ao Curso**

# Aula 0 – Introdução – JavaScript Moderno

Segundo o StackOverflow, em 2019, o JavaScript é a linguagem de programação mais utilizada no mundo.

E quem utiliza essa linguagem?

Grandes empresas como a Google, YouTube, LinkedIn, Netflix, facebook, Uber, PayPal, etc.

Nesse curso vamos aprender os fundamentos da linguagem JavaScript, vamos aprender também a trabalhar no modo gráfico dos navegadores, construções de páginas web interativas utilizando a linguagem e vamos trabalhar também um pouco com NodeJS, que executa o JavaScript fora dos navegadores.

Esse é um curso básico da linguagem, mas também bem moderno pois ele traz a especificação ECMAScript, que é a padronização internacional da linguagem JavaScript.

O curso será composto por seis módulos, com 17 aulas ao total, onde faremos 8 exercícios práticos.

# Módulo A: Conhecendo o JavaScript

Conforme abordado na primeira aula o curso será composto por seis módulos e nesse primeiro teremos a seguinte composição:

* **A** 🡪 Conhecendo o JavaScript
* O que o JavaScript é capaz de fazer?
* Dicas de aprendizagem
* JavaScript vs ECMAScript
* Requisitos de Software?
* Primeiros Scripts em JS

# Aula 1 – O que o JavaScript é capaz de fazer?

Você sabe para que serve a linguagem JavaScript / ECMAScript? Sabe a diferença entre um cliente e um servidor para a Internet? Sabe para que servem as tecnologias HTML5, CSS3 e JavaScript? Consegue citar 4 sites que utilizam a linguagem JavaScript nos seus códigos?

Pois, essas e muitas outras perguntas, serão respondidas ao longo dessa aula. E não podemos nos esquecer sempre de praticar todas as atividades que faremos daqui para frente durante o curso.

A primeira pergunta que geralmente se faz é o que o JavaScript é capaz de fazer?

E a resposta para isso é: **TUDO !!!**

Porém, para começarmos, precisamos entender um conceito muito importante na internet. O conceito de ***Cliente*** e ***Servidor***.

*Clientes*, podem ser, por exemplo, os nossos computadores e/ou celulares que utilizamos normalmente no dia a dia e que utilizam serviços fornecidos por um *Servidor*, como por exemplo, ao assistir um vídeo no YouTube o Cliente está recebendo dados dos servidores do YouTube.

Esses servidores, que são maquinas de alta performance, possuem um sistema operacional e diversas pastas e diretórios internos que fornecem dados para os clientes de forma a atender um serviço, seja ele um vídeo no YouTube, uma mensagem no WhatsApp, uma notícia no Facebook, etc.

Então, no momento em que usamos um dispositivo na internet, esse dispositivo é um Cliente, que precisa de dados de um Servidor(es).

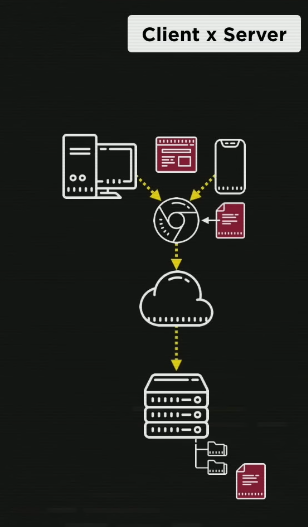
E como fazemos esse acesso na internet?

No momento em que usamos um navegador e fazemos uma pesquisa de um site ou URL que queremos buscar, esse nome ou URL que digitamos para fazer a pesquisa vai até a infraestrutura da internet e ela vai responder exatamente para qual servidor seremos que desviados.

O dados fornecidos por um servidor, contidos em suas pastas e diretórios, podendo ser uma imagem, vídeo, e-mail, etc. na maioria das vezes é um arquivo HTML. E quando solicitamos esse arquivo o servidor nos envia uma cópia diretamente para o Cliente, ou seja, em nosso navegador. A partir daí os navegadores irão trabalhar para apresentar esse arquivo e gerar o efeito visual e gerar o site, da maneira que queremos.

Dentro desses conceitos, o JavaScript surgiu como uma tecnologia voltada para os Clientes, que será a parte em que iremos focar nesse curso.

Obs.: Nos dias atuais o JavaScript funciona também nos servidores.



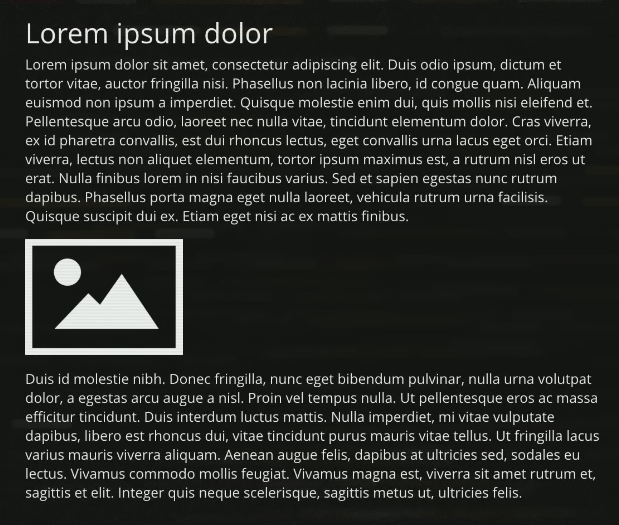
Dessa forma, para nós, podemos entender o JavaScript como uma tecnologia ***ClientSide***.

Um outro fator que é importante se entender junto ao JavaScript são as tecnologias relacionadas.

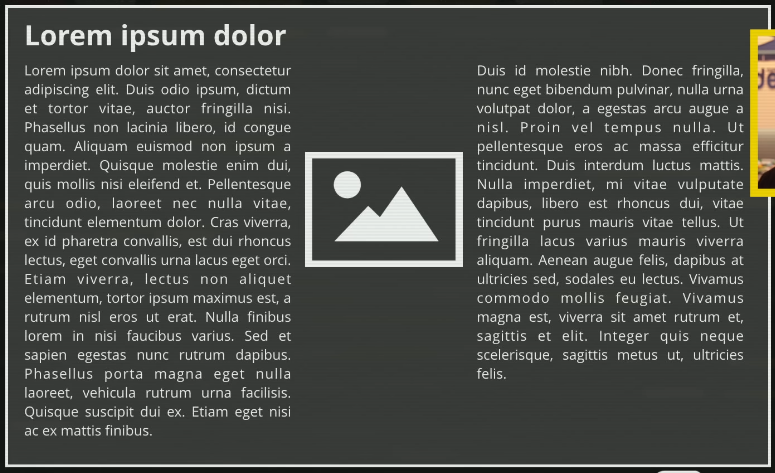
Um site, como o que citamos para exemplificar a relação de cliente x servidor, na grande maioria das vezes é composto por 3 tecnologias do lado do cliente. São elas: **HTML**, **CSS** e **JavaScript**.

Para exemplificar cada uma delas, imagine um jornal, onde se necessitam algumas funções que precisam ser exercidas para se ter aquele produto final ao consumidor.

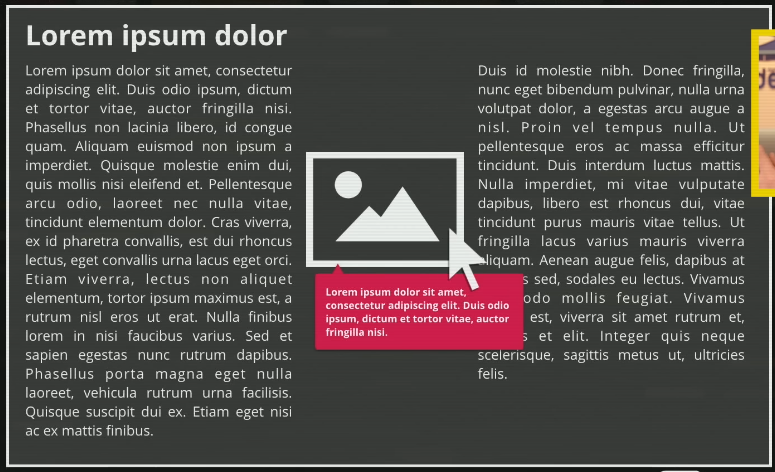
Uma dessas funções é a jornalista, que precisa escrever o texto, buscar imagens para apresentar a matéria ou em alguns casos vídeos para ilustrar aquilo que se estar explicando. Em outras palavras essas jornalista seria a responsável pelo conteúdo do Jornal. Essa função seria a do HTML.



Outra função seria a do designer que pegara o conteúdo, imagens e textos preparados pela jornalista e vai tornar aqui muito mais bonito e vistosos os olhos do consumidor, dimensionando e posicionando o conteúdo do jornal. Essa função é a das CSS.



Programador seria o responsável pela engenharia do jornal, ou seja, ele vai pensar em que áreas precisamos entregar mais jornal, vai se preocupar com as integrações entre, por exemplo, a empresa que produz e que entrega esse jornal, além da segurança. Essa função é a do JavaScript.



O JavaScript, em outras palavras, é responsável pela interação que o usuário tem com o site. Ele permite fazer quase tudo, até mesmo modificar o documento na sua parte HTML e CSS, esse é o poder da linguagem.

Conforme já falamos ele é utilizado nas maiores empresas do mundo como a Google, YouTube, LinkedIn, Netflix, facebook, Uber, PayPal e até mesmo em sites do governo como o do Brasil e do EUA.

Até aqui, com os conceitos e exemplos que vimos, conseguimos ter um pouco da noção do poder que a linguagem JavaScript possui.

E a partir daqui começaremos a aprender um pouco mais dos fundamentos nas próximas aulas.

# Aula 2 – Como chegamos até aqui?

Para entendermos até onde a linguagem JavaScript pode nos levar precisamos primeiro entender de onde ela veio.

A história do JavaScript surge com a história da própria internet, em 1970, quando surgiu os primórdios da internet, no período da guerra fria.

Nesse período foi criada uma agencia de pesquisas tecnológicas chamada DARPA, que tinha como função pesquisar novas tecnologias para a guerra. Uma dessas tecnologias envolvia a segurança dos centros militares, que tinham como principal função a proteção dos dados que circulavam e eram armazenados nesses centros, e como durante esse período poderiam ser facilmente bombardeados precisava-se de uma forma de garantir a segurança, principalmente, dos dados.

Nesse sentido foi criada a *Arpanet*, que era uma rede da internet que se comunicava com outros centros militares, dessa forma se algum dos centros fossem destruídos durante a guerra a informação contida nele e nas pesquisas não se perderiam, pois continuariam tendo acesso a ela através de outros centros militares, os danos então, seriam totalmente físicos.

Essa rede era gerenciada principalmente pelo governo americano e pelos militares, porém haviam também grandes universidades dentro desse projeto, como a Universidade da Califórnia e o próprio MIT.

Essa rede chamada *Arpanet* com o tempo foi se expandindo de forma tão grande que o próprio governo não estava mais dando conta e a ela se juntaram outras pequenas redes. E foi nesse momento que a *Arpanet* mudou de nome algumas vezes sendo que o último deles dado a ela foi o de Internet. Sim, a Internet que conhecemos hoje!

Com o passar dos anos essa evolução deu origem a diversas tecnologias, sendo uma delas muito importante criada em 1993, em Genebra, por um inglês chamado Timothy B. Lee, que pesquisou um meio de transformar o conteúdo em algo mais interativo através de ligações entre documentos, dando origem assim a linguagem HTML, o protocolo *http* e fundando a World Wide Web (WWW).

Para que essas descobertas e pesquisas de Timothy B. Lee funcionassem se fazia necessário um navegador, surgia então nos EUA o Mosaic, que foi a invenção de Mark Andreessen, um pesquisador do NCSA (Centro nacional de aplicação de supercomputadores).

O Mosaic foi o primeiro navegador da história, adaptado de um interpretador do protocolo anterior ao *http* e que também mostrava páginas web.

Mark Andreessen também foi muito importante em 1994 quando ele saiu do NCSA e se juntou a um bilionário e fundador da Silicon Graphics, chamado Jim Clark. Juntos os dois criaram uma empresa chamada NETSCAPE que desenvolveu um navegador baseado no Mosaic, criação anterior de Mark Andreessen.

Com o crescimento da NETSCAPE e baseando-se na necessidade de evoluir o HTML, que era muito estático, Brandon Eich, um desenvolvedor e ex funcionário da Silicon Graphics foi contratado pela NETSCAPE com o objetivo de criar uma nova linguagem que desse mais funcionalidades ao simples HTML da época. O Brandon Eich batizou a sua linguagem inicialmente de Mocca e mais pra frente naquele mesmo ano, veio a ser conhecida como LiveScript e nessa mesma época estava surgindo uma outra linguagem da Sun Microsystems chamada de JAVA, que além de ficar famosa muito rápido passou a ser conhecida por todos do meio como a linguagem do futuro.

Dessa forma, a NETSCAPE e o Brandon Andreessen, tendo em vista a fama da linguagem Java e enxergando uma possibilidade de trazer um pouco disso para sua nova linguagem mudou o seu nome para a linguagem que conhecemos hoje em dia, o JavaScript.

Muitos acham que a linguagem JavaScript tem tecnicamente algo a ver e se parece com a linguagem Java, ou que seria alguma derivação dela, porem como pudemos ver não se tratava disso, mas sim de uma grande jogada de marketing pois na época o Java era a linguagem mais falada no mundo.

Com o sucesso que o JavaScript foi tendo com o tempo isso atraiu os olhos de grandes empresas para a linguagem, como a própria Microsoft, que tinha acabado de lançar o seu próprio navegador baseado no código fonte do Mosaic, chamado Internet Explorer.

Em 1997, temendo que outras empresas como a própria Microsoft, que tinha um projeto de criar uma linguagem para o seu navegador baseado no JavaScript que seria chamado de JScript, a NETSCAPE teve uma grande sacada, que foi padronizar a sua linguagem procurando uma grande empresa europeia que é a ECMA (Associação Europeia de fabricantes de computadores).

A ECMA na Europa é como a ISO nos EUA, ela padroniza as coisas e com a NETSCAPE fornecendo o código da sua linguagem para essa associação que surgiu o ECMAScript. Basicamente o ECMAScript é a própria linguagem JavaScript padronizada.

Avançando para 2002 onde é dado o fim da ‘treta’ entre a Microsoft e a NETSCAPE, que por sua vez deixou de ter o navegador mais utilizado do mercado, pois o Windows 98 da Microsoft já vinha com um navegador próprio embutido, que é o Internet Explorer, e com isso a NETSCAPE acabou falindo nesse ano.

Ainda em 2002, os ex funcionários e desenvolvedores da NETSCAPE, ainda empenhados em desenvolver um novo navegador, se juntaram, após a sua falência, e montaram uma fundação que ganhou o nome de um de seus projetos que estavam sendo desenvolvidos ainda na época da antiga empresa, o Mozilla. Após algum tempo esse projeto ganhou um novo modelo, um novo código e uma nova roupagem e teve seu navegador reconhecido como Firefox.

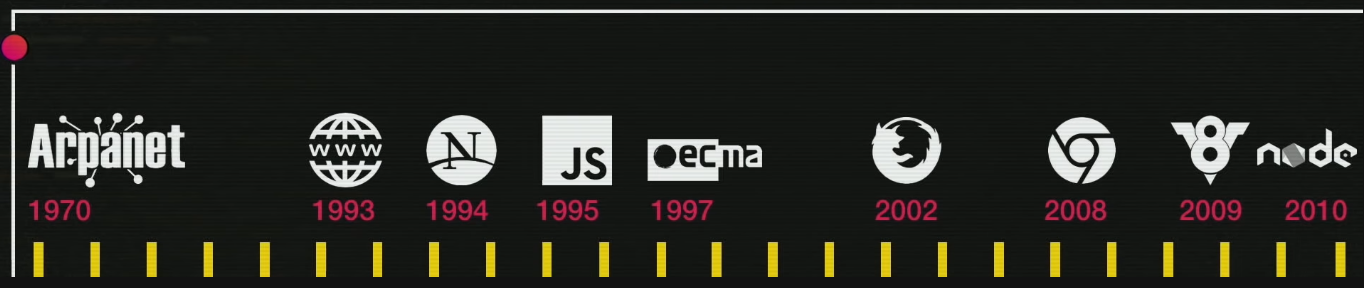
O Firefox foi um navegador muito querido e popular por muitos anos, até que em 2008, com o crescimento e avanço do mercado da internet cada vez maior, uma grande empresa também estabeleceu um projeto para criar o seu próprio navegador, surgindo então, o Google Chrome, desenvolvido pela Google, que em menos de 3 anos se tornou o navegador mais usado no mundo e até os dias de hoje ele detêm cerca de 50% da utilização entre todos os navegadores do mundo.

O Google Chrome trouxe consigo coisas muito interessantes principalmente para o JavaScript. Ele possui um motor interno de JavaScript muito poderoso chamado versão V8 (uma referência aos motores automobilísticos), que surgiu em 2009.

Cada navegador possui o seu motor e uma das grandes vantagens desse motor do Google Chrome, e que tornou ele muito importante, é que ele gerava código JIT (Just in time), além de ser de código aberto. E foi através desse último fator que um grupo de pessoas pegou esse código livre, mexeu e realizou alterações, fazendo com que o motor V8 do navegador funcionasse fora do Google Chrome e essa iniciativa deu origem a uma famosa ferramenta em 2010, o NodeJS.

**O NodeJS é uma máquina que roda JavaScript fora do navegador.** Isso possibilitou que o JavaScript pudesse ser trabalhado não mais somente pelo lado do Cliente mas agora também pelo lado do Servidor.

Linha do tempo dos acontecimentos importantes para o JavaScript.



O ECMAScript também teve a sua evolução. Como vimos anteriormente, o ECMAScript é a versão padronizada do JavaScript, surgindo em 1997 em sua versão 1.0 utilizando o JavaScript 1.1 que foi padronizado pela ECMA.

Mais tarde em 1998 foi lançada a versão 2.0 que trouxe apenas algumas poucas atualizações.

Em 1999, veio a versão 3.0, que trouxe compatibilidade com expressões regulares e bloco para tratamento de erros. Nesse ano a evolução para uma versão 4.0 ficaram um pouco travada, pois as promessas de evolução eram tão grandes que a versão nunca saia, o que acabou de fato acontecendo, pois nunca se teve uma versão 4.0.

Muitos anos depois, em 2009, pulando a versão 4, surgiu o ECMAScript versão 5.0 que ficou conhecido como ES05, sendo essa a versão mais popular de todas, compatível com praticamente todos os navegadores atuais. Ela se tornou compatível com JSON (que é uma formatação leve de troca de dados. Para seres humanos, é fácil de ler e escrever.) e trouxe vários métodos para tratar *arrays* (que são estruturas que servem para guardar dados, e organizá-los.).

Em 2015 surgiu o ES06 que trouxe os declaradores *let* e *const* e também as *strings* em templates.

A partir do ES06 surgiu uma versão do ECMAScript por ano que passaram a ser conhecidas por ES + o ano da versão, como o ES2016 que trouxe algumas funcionalidades como, por exemplo, o operador de exponenciação.

O ES2017 trouxe compatibilidade com funções assíncronas entre outras muitas coisas.

O ES2018 trouxe novidades em RegEx (Expressões regulares) e também trouxe a possibilidade de *promisses*.

Com o tempo surgiram novas tecnologias, como o próprio NodeJS, JQuery, Angular (Google), React (Facebook) e Vue.JS que levam o JavaScript para outro patamar.

Sendo eles:

**JQuery** – Um conjunto de bibliotecas. Foi muito famoso por muitos anos apesar de uma queda em sua utilização recentemente e ele foi criado pelo time do Mozilla, trazendo uma biblioteca que ajudava e facilitava muito em interatividades do JavaScript;

**Angular** – Surgido em 2009, criado e mantido pelo Google, e ele facilita muito a criação de aplicações web. Ele traz uma linguagem menos imperativa e mais declarativa, se assemelhando muito ao SQL para banco de dados. Existem variações entre o Angular 1.0 (tradicional), o AngularJS e o Angular (a partir da versão 2).

**React** – Famosa biblioteca lançada pelo Facebook, o React traz a mesma funcionalidade do Angular porém com algumas flexibilidades a mais como o React Native, para dispositivos mobile.

**Vue.JS** – Surgiu em 2014 criado por um ex programador do Google que, segundo ele, estava cansado de usar o Angular, traz uma versão muito forte e muito popular de um framework JavaScript de código-aberto, focado no desenvolvimento de interfaces de usuário e aplicativos de página única.

**Electron** – Ferramenta mantida pelo GitHub especializada em criação de interfaces gráficas. O VSCode foi feito em Electron, assim como o app do WhatsApp para desktop e o chat do discord.

**Ionic** – é um SDK (Kit de desenvolvimento de software) de código aberto completo para desenvolvimento de aplicativo móvel híbrido. O Ionic assim como vários dessa lista, utiliza o NodeJS para rodar o JavaScript fora do navegador.

Obs.: *SDK é tipicamente um conjunto de ferramentas de*[*desenvolvimento de software*](https://pt.wikipedia.org/wiki/Desenvolvimento_de_software)*que permite a criação de*[*aplicativos*](https://pt.wikipedia.org/wiki/Software_aplicativo)*para um certo pacote de software,*[*framework*](https://pt.wikipedia.org/wiki/Framework)*, plataforma de hardware,*[*sistema de computador*](https://pt.wikipedia.org/w/index.php?title=Sistema_de_computador&action=edit&redlink=1)*,*[*console de videogame*](https://pt.wikipedia.org/wiki/Console_de_videogame)*,*[*sistema operacional*](https://pt.wikipedia.org/wiki/Sistema_operacional)*, ou plataforma de desenvolvimento similar.*

# Aula 3 – Dando os primeiros passos

Nesta aula vamos começar a dar os primeiros passos na linguagem JavaScript.

Mas antes de começarmos é sempre importante relembrar, aprender programação é **PRATICA** !!!! Não basta apenas leitura, assistir vídeos, etc. é preciso colocar a mão na massa.

1. Mas como aprender?

- Ler bons livros, manuais e referencias;

- Assistir muitos vídeos;

- Praticar SEMPRE;

- Anotar tudo;

- Interações com outras pessoas;

- Construir projetos.

Toda ajuda e conteúdo é sempre muito valido e importante para aprender programação!

2. Bibliografia recomendada:

*JavaScript – O guia definitivo. David Flanegan, Editora: O`REILLY.*

*JavaScript – O guia do programador. Mauricio Samy Silva.*

3. Guias e Referencias:

3.1. JavaScript

Guia: <https://developer.mozilla.org/pt-BR/docs/Learn/JavaScript> (PT-br)

Referência: <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference> (PT-br)

3.2. ECMA

Referência: https://ecma-international.org/publications-and-standards/standards/ecma-262/ (EN)

4. Respondendo dúvidas comuns

- Não existe idade mínima para aprender e trabalhar com programação.

- Não existe gênero para aprender e trabalhar com programação.

- Não precisa morar em grandes centros para aprender e trabalhar com programação.

- Não precisa ser expert em matemática para aprender e trabalhar com programação.

- Não precisa saber falar inglês para aprender e trabalhar com programação.

- Aprender lógica de programação é muito importante para aprender e trabalhar com programação.

- Não existe uma linguagem melhor do que a outra, cada linguagem tem o seu propósito.

- Não tenha pressa, comece programando coisas simples, do básico.

Agora que já esclarecemos esses pontos, vamos aos primeiros passos!

5. Instalações

Para o nosso curso vamos precisar de um navegador instalado, recomenda-se o **Google Chrome**, pois possui algumas ferramentas que vão auxiliar no aprendizado.

Um editor de código, recomenda-se o **VSCode**.

E por fim, o **NodeJS**, pois como já aprendemos ele irá ajudar a rodar o JavaScript fora do navegador.

# Aula 4 – Criando o seu primeiro script

Agora que temos tudo que precisamos instalada chegou a hora de colocar a mão na massa e executar nosso primeiro código em JavaScript.

Como falamos anteriormente, um site ele tem sua base em 3 fatores:

O primeiro a sua estrutura através de **HTML**, o segundo através dos estilos com **CSS** e terceiro as suas interações com **JavaScript**.

Então levando como base essa estrutura, em que momento inserimos o JavaScript?

O JavaScript normalmente é inserido por último, ao fim da tag *<body>*, bem antes do seu fechamento. Isso pois o ideal é que os *scripts* sejam carregados após o código base.

Para inserir um código em JavaScript em nosso site no documento HTML precisamos inserir a tag ***<script>.*** Essa tag possui abertura e fechamento e é dentro dela que criamos e programamos os nossos scripts locais que irão rodar em nossos sites.

Ex01: Criando o primeiro comando em JavaScript

Vamos tomar como base uma página simples, com um título e um parágrafo e um estilo local básico, conforme abaixo:

**<title>Meu primeiro programa...</title>**

**<style>**

**body {**

**background-color: rgba(0, 0, 255, 0.356);**

**color: azure;**

**font: normal 20pt Arial;**

**padding-left: 10px;**

**}**

**</style>**

**</head>**

**<body>**

**<h1>Olá, mundo !</h1>**

**<p>Já me livrei da maldição</p>**

**</body>**

Resultado:



A partir dela vamos criar nosso primeiro comando em JavaScript para criar um alerta que será exibido na tela após carregarmos essa página.

Para isso vamos inserir ao final do nosso código a tag ***<Script>*** e dentro dela inserir o seguinte comando:

**<body>**

**<h1>Olá, mundo !</h1>**

**<p>Já me livrei da maldição</p>**

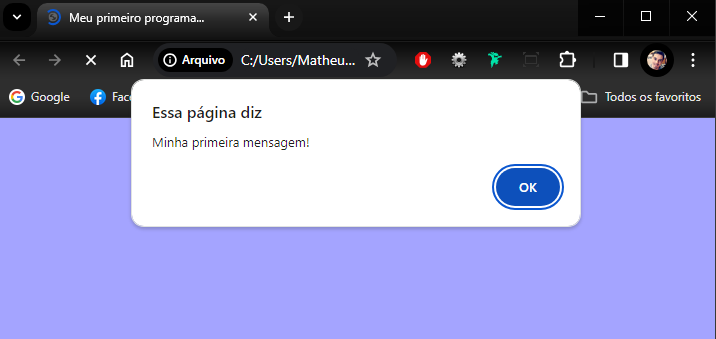
**<script>**

**window.alert('Minha primeira mensagem!')**

**</script>**

**</body>**

Resultado:



O comando **window.alert()** cria uma mensagem de alerta na tela para que seja exibida toda vez que acesse a página ou que ela seja carregada.

Dentro dos parênteses a mensagem que queremos exibir deverá estar sempre entre aspas simples, conforme acima.

Note que o comando foi escrito ao final do código, após as partes que envolvem o HTML e o CSS, porém ao carregar a página notamos que o primeiro a rodar é o JavaScript apresentando a mensagem na tela, isso é a nossa parte de interatividade.

Em JavaScript a maioria dos comandos são escritos em letras minúsculas, porém há alguns casos em que a letra maiúscula se é necessária para diferenciações. Outro ponto importante é que nos códigos em JavaScript, diferentemente do CSS, PHP, Java, etc., o uso de ***;*** (ponto e vírgula) não é obrigatório, então é possível escrever todo o código de forma simples sem o uso do sinal.

Obs.: Há pessoas que dizem ser mais correto ou padronizado utilizar o ponto e vírgula ao final dos comandos, porém o fato é que isso não é uma regra em JavaScript.

Agora que já aprendemos o primeiro comando vamos adicionar mais dois a essa mesma página.

**<script>**

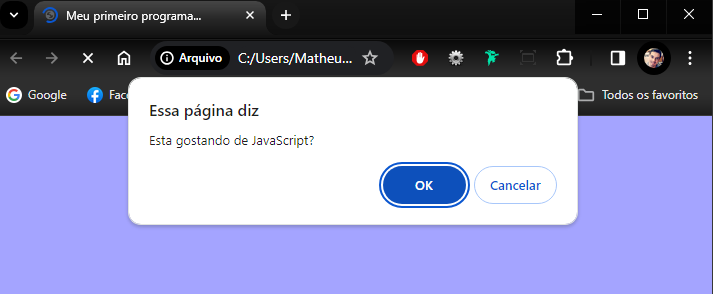
**window.alert('Minha primeira mensagem!')**

**window.confirm('Esta gostando de JavaScript?')**

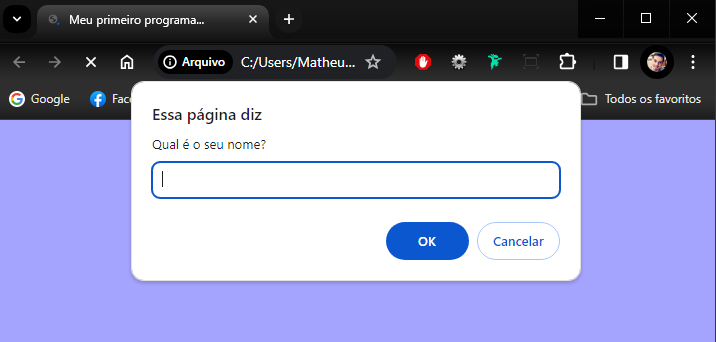
**window.prompt('Qual é o seu nome?')**

**</script>**

Através do **window.confirm** será exibida uma nova mensagem na janela com o texto **'Esta gostando de JavaScript?'** e uma opção de confirmação. Conforme abaixo.



Através do comando **window.prompt** será exibida uma nova mensagem na janela com o texto **'Qual é o seu nome?'** e um campo para preenchimento do nome. Conforme abaixo.



Note que as mensagens são exibidas uma após a outra de acordo com a ordem dos comandos presentes no código.

Esses são comandos simples sem grandes validações ou regras mas que mostram um pouco da interatividade que o JS pode nos trazer.

Agora, se nos perguntarmos se é possível preenchermos o nosso nome nessa última mensagem e ele ser exibido em tela, a resposta é sim! Porem para isso será necessário mais alguns comandos de código e inserirmos mais alguns elementos que vamos aprender em JS, como por exemplo, as variáveis.

E são justamente elas que aprenderemos na próxima aula.

# Módulo B: Comandos Básicos do JavaScript

No módulo A aprendemos um pouco mais sobre JavaScript como sua origem, história, curiosidades e avanços, até realizarmos nosso primeiro comando, a partir deste segundo modulo vamos aprender os comandos básicos mais utilizados na linguagem.

* **B** 🡪 Comandos Básicos do JavaScript
* Armazenando dados
* Tratamento de dados
* Operações com dados

# Aula 5 – Variáveis e Tipos Primitivos

Dando sequencia ao nosso estudo em JavaScript, nesta aula vamos aprender um pouco mais sobre variáveis e sobre alguns tipos de dados.

Na aula anterior, aprendemos como disparar alguns alertas na tela como um alerta simples, uma confirmação e um prompt para preenchimento. Mas se pararmos para observar para onde esses dados foram, eles neste formato, se perderam por não especificarmos como armazenar esses dados ou como exibi-los em tela, e são, também, esses e outros pontos que aprenderemos a partir dessa aula.

Em primeiro lugar precisamos destacar a importância dos comentários em um código, pois são através deles que um código pode ser bem documentado, seja para você mesmo caso precise reler ou realizar alguma manutenção nesse mesmo código anos depois ou para outro desenvolvedor.

Em JS adicionamos um comentário no código através de **//** (duas barras) ou **/\* \*/** (como em CSS).

1. **//** 🡪 Referencia um comentário em uma única linha;

Ex01: **<script>**

**//window.alert('Minha primeira mensagem!')**

**//window.confirm('Esta gostando de JavaScript?')**

**window.prompt('Qual é o seu nome?')**

**</script>**

Nesse caso as duas primeiras linhas de código para os dois primeiros alertas viram comentários e não irão rodar quando abrirmos a página.

Uma outra utilização para o / / e ao final da linha de código, neste caso não para inutilizar o comando, mas sim para de fato adicionar um comentário, como por exemplo, uma explicação do que aquela linha de código faz.

Ex01.1: **<script>**

**window.alert('Minha primeira mensagem!')**

**window.confirm('Esta gostando de JavaScript?') //Janela com botao Ok e Cancelar**

**window.prompt('Qual é o seu nome?') //Alerta para perguntar seu nome...**

**</script>**

1. **/\* \*/** 🡪 Referencia um comentário em mais de uma linha de código.

Ex02: **<script>**

**/\***

**window.alert('Minha primeira mensagem!')**

**window.confirm('Esta gostando de JavaScript?')**

**\*/**

**window.prompt('Qual é o seu nome?')**

**</script>**

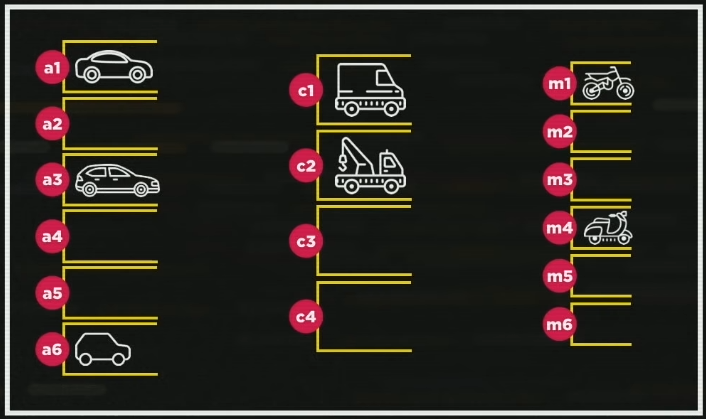
Obs.: Use-os com moderação! Apesar de ter um código comentado e explicado ser importante, há pessoas que comentam absolutamente tudo e sem necessidade e isso acaba tornando o código extremamente poluído.

**5.1 – Variáveis**

As variáveis ​​são parte integrante de quase todas as linguagens de programação e geralmente são um dos primeiros tópicos que aprendemos quando começamos a codificar.

Variáveis ​​são usadas para armazenar valores de dados. Por exemplo, uma variável pode ser usada para armazenar o endereço de e-mail de um usuário ou seu nome. Em JavaScript, uma variável pode conter qualquer tipo de dado.

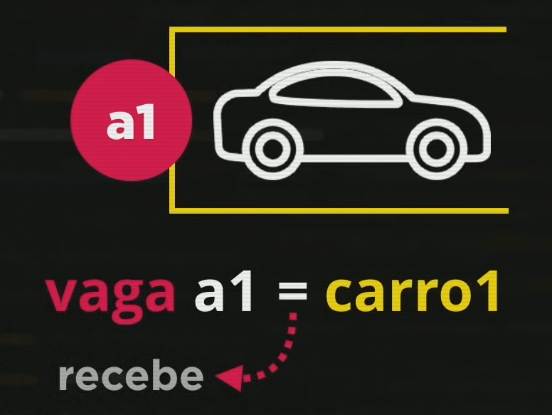
Veja com um exemplo prático, imagine um estacionamento com vagas para carros, caminhões e motos. Conforme imagem abaixo.



Nesse caso as vagas seriam as nossas variáveis. Ou seja, para cada tipo de automóvel existe um tipo de vaga especifica para atende-la.

E como identificamos essas vagas?

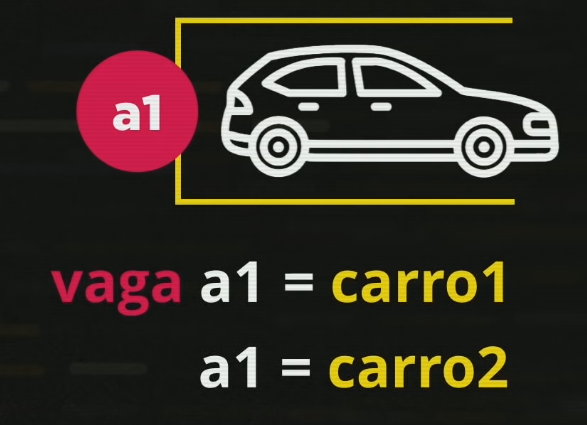
Para cada variável existem o que chamamos em JS de identificadores. Esses identificadores são nada mais nada menos que um nome ou, como no exemplo abaixo, um código (a1) que vai identificar aquela variável (vaga).



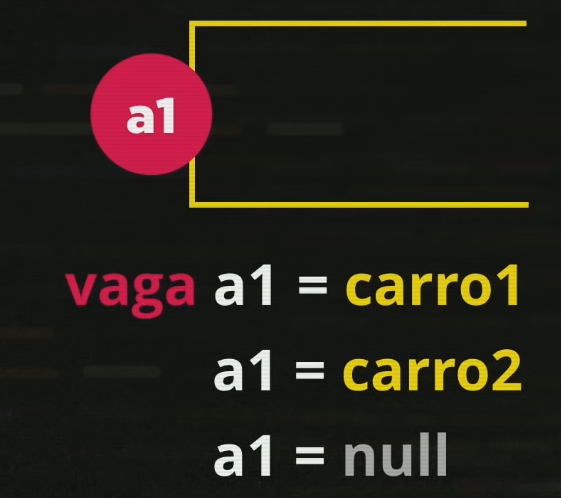
Um ponto importante para fixar e que em JS o sinal de **=** significa **‘receber’** em uma declaração. Ou seja, na variável acima estamos declarando que a vaga de identificação a1 está recebendo (=) o carro 1.

* Variável: **vaga**
* Identificador: **a1**
* Valor: **carro1**

Outro ponto que precisamos reforçar e’ que se a minha variável já esta preenchida com um valor ela não poderá receber outro. Retornando ao exemplo pratico do estacionamento fica ainda mais claro, pois se eu já tenho um carro parado em uma determinada vaga, fica impossível que outro estacione ali.



Por fim, um outro valor que essa variável pode receber e’ o valor de ***null***, ou seja, nulo, onde essa vaga (variável) ficara vazia.

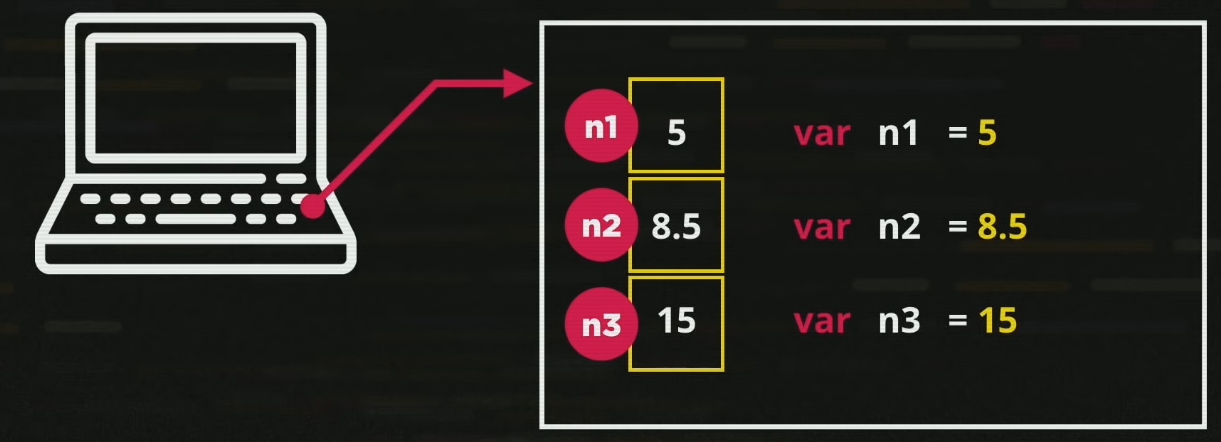


Agora que entendemos como esse exemplo prático funciona vamos trazer isso para o computador, pois assim como o estacionamento, o computador possui um terreno, que chamamos de memória onde podemos colocar dados. E assim como o estacionamento precisamos de espaços delimitados para inserir esses dados, as **variáveis**.

IMPORTANTE: Em JS utilizamos a palavra **var** pra declarar uma variável.

Obs.: No JS moderno utilizamos também a palavra **let**.

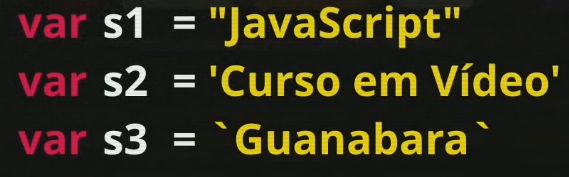
Nesse novo exemplo abaixo podemos ver como funciona essa relação de variáveis em um computador normalmente.



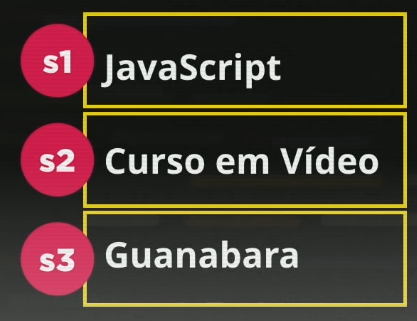
Onde declarando a minha variável **var** de identificador **n1** que recebe o valor **5**, podemos visualizar o “terreno” referente a variável n1 sendo preenchido pelo valor recebido.

Assim como ocorreu no exemplo dos estacionamentos, existem também variáveis de tamanhos diferentes para caber dados de tamanhos diferentes. Como por exemplo abaixo.

Podemos, por exemplo, inserir palavras como dados em nossas variáveis, o que chamamos em JS de ***strings***.

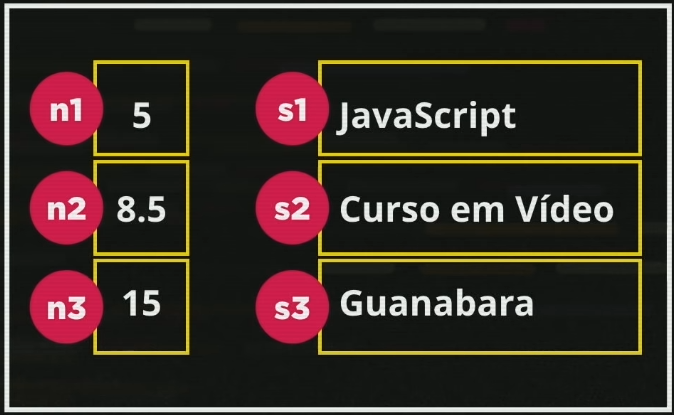


E nesse caso, teremos:



Obs.: Observe que essas palavras (dados) estão entre aspas, e em JS pode-se usar 3 tipos de aspas: aspas duplas, aspas simples e a crase. ISSO TEM DIFERENCA DENTRO DO JAVASCRIPT! São três diferentes formas de se delimitar uma string. Porem vamos falar melhor disso um pouco mais para frente.

Sendo assim, ao final podemos concluir nesse exemplo que temos 6 espaços delimitados e bem definidos, onde cada um possui o seu nome (identificador).



5.2 – Identificadores

Agora que já conhecemos um pouco sobre as variáveis, vamos nos aprofundar um pouco mais sobre os identificadores e a primeira coisa que precisamos saber é que existem algumas regras para nomeação de identificadores. Não podemos simplesmente sair colocando qualquer nome.

**Regra 1 – Podem começar com letra, $ ou \_**

**Regra 2 – Não podem começar com números**

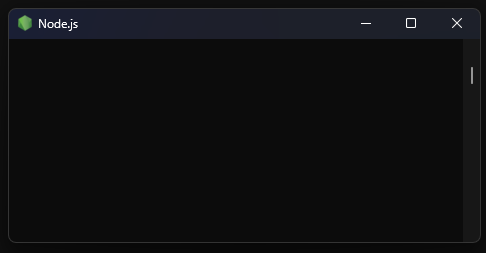
**Regra 3 – É possível utilizar letras e números**

**Regra 4 – É possível usar acentos e símbolos**

**Regra 5 – Não podem conter espaço (normalmente se separa com \_ )**

**Regra 6 – Não podem ser palavras reservadas (que já possuem outras atribuições. Ex: function, alert, var, etc.)**

Para colocarmos um pouco mais em pratica esses conceitos aprendidos, vamos utilizar o NodeJS para alguns exemplos. Para isso podemos abrir o NodeJS de algumas maneiras, a primeira dela é através do próprio app que baixamos anteriormente, onde ele abrirá uma interface similar a um terminal cmd.

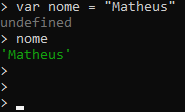


Ao abrir a interface do NodeJS podemos realizar alguns comandos como uma simples operação de soma, subtração ou até mesmo declarar uma variável assim como aprendemos anteriormente:



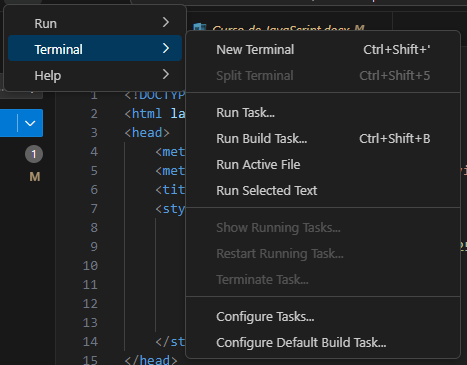
Onde temos uma variável de identificador nome que recebe o valor em string “Matheus”.

Note que ao dar ENTER o NodeJS responde como ‘undefined’, isso pois não pedimos a ele que exibisse o nome na tela. Mas uma vez que solicitamos pelo nome da variável que criamos temos no seguinte resultado:

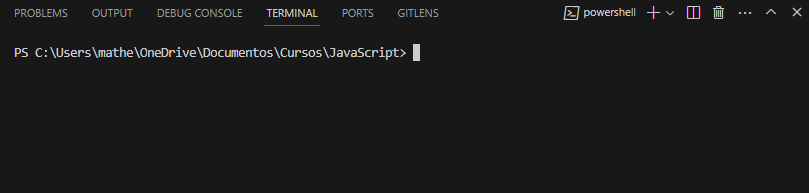


Para sair do prompt do Node digitamos ***.exit***

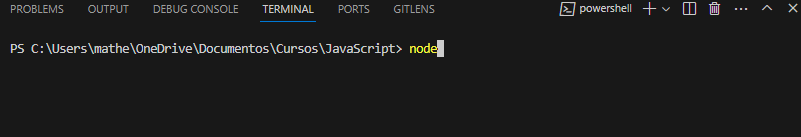
Outra forma de acessar o NodeJS é no próprio VSCode, através da sessão ‘Terminal’ na opção ‘New Terminal’ ou com o comando CTRL+SHIFT+’. Conforme abaixo.



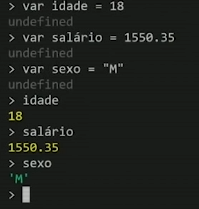
Com isso será aberto um terminal dentro do VSCode, conforme abaixo.



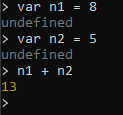
E para acessarmos o node por ali basta digitarmos nesse terminal: ***node***.



E conseguimos declarar as variáveis da mesma forma por aqui



Através dessas variáveis conseguimos realizar operações simples como uma soma. Declarando duas variáveis e após solicitando que os dois identificadores sejam somados.



Caso a tela no Node fique muito cheia de comandos podemos também limpa-la com o comando CTRL+L

Mesmo seguindo todas essas regras citadas acima, alguns desenvolvedores deslizam na hora de criar os nomes para os identificadores. Então existem também algumas dicas para se utilizar nesses momentos.

**Dica 1 – Letras maiúsculas e minúsculas fazem diferença. Precisa-se ter muita atenção na hora de se declarar a variável pois uma simples mudança de letra na hora de chama-la resultara no não funcionamento.**

**Dica 2 – Tente escolher nomes coerentes para a variável que se está criando.**

**Dica 3 – Evite se tornar o ‘programador alfabeto’, onde todas as variáveis que usa são a,b,c,d,e,f... ou ‘programador contador’, onde todas as variáveis que usa são n1, n2, n3, n4... .**

Por fim, variáveis servem nada mais nada menos para se guardar dados, e diferentes tipos de dados.

Um dado pode ser um número, uma palavra, um valor único, etc. Por isso existem algumas definições que utilizamos para alguns dos tipos de dados. E dentre esses tipos temos os três tipos primitivos primordiais: ***Number***, ***String*** e ***Boolean***.

* **Number**: são números, do tipo inteiro (*integer*), sem casa decimal após a virgula e sem parte fracionaria, e os reais (*float*) com ponto flutuante. Em JS todos esses tipos são apenas um, o *number*.



* **String**: são cadeias de caracteres, como palavras.

*Obs.: Uma string também pode ser um conjunto de números, como por exemplo, um telefone, um CPF, um RG. Todos esses são números mais possuem pontos e traços na sua composição, então não são totalmente numéricos, nesses casos se definem como string.*



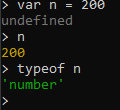
* **Boolean**: são os valores de verdadeiro e falso, valores do tipo boleano.



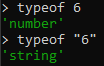
É valido lembrar que em JS existem outros tipos de dados, como dentro do tipo *number* podemos encontrar os tipos ***Infinity*** e ***NaN***(Not a Number). Existem também valores como ***Null*** e ***Undefined***. Temos também o ***Object*** (como JS é uma linguagem orientada a objetos) que internamente possui o ***Array*** (Vetores). E o tipo ***Function***.

E para trabalhar com todos esses tipos existe um comando em JS muito valioso, o **typeof** (Tipo de).

Através dele conseguimos saber o tipo de dado que esta sendo trabalhado naquela variável, conforme abaixo.



Lembrando que se eu coloco qualquer valor entre aspas, seja ele até mesmo um número, esse valor será uma *string*.



# Aula 6 – Tratamento de dados

Dando continuidade ao que aprendemos na aula anterior vamos começar a trabalhar a manipulação de dados em JS.

Se retomarmos o primeiro exercício do curso, o *ex001.html*, criamos nele alguns comandos de alerta que aparecem na tela ao carregar a página, porém ao clicarmos, preenchermos nosso nome e interagirmos com esses alertas, não temos nenhum resultado pratico ou visual, isso porque como comentamos no exercício, precisamos de mais algumas linhas de código para exibir esses resultados.

Para isso vamos utilizar as variáveis, onde vamos inserir dentro de uma delas o resultado desse preenchimento no alerta.

Ex01: Exibindo nome da tela com variáveis.

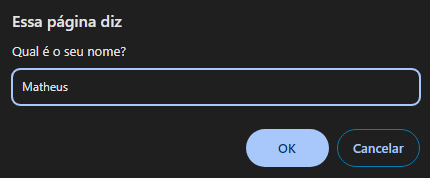
Para esse caso precisamos que o nome que for preenchido no prompt do alerta seja gravado em algum lugar para que depois possamos exibi-lo. Sendo assim vamos criar uma variável chamada *nome*, para gravar esse nome inserido no alerta, conforme abaixo.

**<script>**

**var nome = window.prompt('Qual é o seu nome?')**

**</script>**

Dessa forma, o nome que preenchermos no alerta será gravado dentro da variável de identificador nome que está recebendo o alerta de preenchimento.



Mas como faremos para que ele seja exibido?

Vamos criar um novo alerta na pagina para exibir essa nova variável, conforme abaixo.

**<script>**

**var nome = window.prompt('Qual é o seu nome?')**

**window.alert('É um grande prazer te conhecer, ' + nome)**

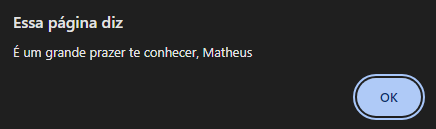
**</script>**

Nela estamos criando um alerta que exibirá a mensagem entre aspas e adicionará ao final o resultado gravado na nossa variável (**nome**).

Em JS quando criamos uma variável podemos utiliza-la de diversas maneiras em nosso código mais a frente e, a partir da criação, podemos passar a utilizar simplesmente o seu identificador para rodar os dados inseridos, como fizemos acima.

O sinal de **+** tem a função de concatenar o alerta ao resultado que queremos, o valor da variável **nome**.

Dessa forma, teremos como resultado:



Obs.: Por isso devemos ter muito cuidado ao dar nomes as variáveis, conforme falamos anteriormente, como utilizaremos elas ao longo de um desenvolvimento, podemos acabar atribuindo a ela uma funcionalidade de um elemento que já existe. Por isso é muito importante seguir as regras e dicas citadas na aula anterior.

Ex03: Conversão de dados

Dando continuidade a um novo exercício vamos agora trabalhar com alguns números para que sejam exibidos na tela e também para que apresente a sua soma.

Para isso, vamos precisar criar 3 variáveis, sendo duas para que o usuário insira um número através de um prompt de alerta e a terceira para gerar uma soma desses dois números e, ao final, vamos inserir um alerta para que essa soma seja exibida em tela.

**var n1 = window.prompt('Digite um número:')**

**var n2 = window.prompt('Digite um segundo número:')**

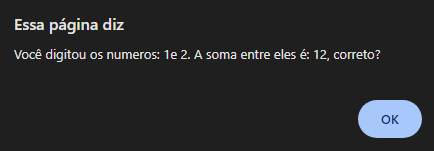
**var soma = n1 + n2**

**window.alert('Você digitou os números: ' + n1 + 'e ' + n2 +'. A soma entre eles é: ' + soma +', correto?')**

Sendo assim, teremos a seguinte sequência de resultados:







Note que os números que inserimos são corretamente apresentados ao final, porem a sua soma não é o resultado que queremos.

Isso porque o sinal de **+** em JS tem funcionalidade dupla, ele serve tanto para concatenar, como feito no resultado acima, quanto para adição e o próprio JS confunde isso, ele não sabe de imediato qual a intenção do resultado que queremos, sendo assim ele segue a seguinte regra: Se os valores informados forem do tipo ***number*** ele realiza uma operação de adição. Se os valores informados forem do tipo ***string***, ele os concatena.

O comando **window.prompt**, independente se o valor preenchido for uma palavra ou um número, ele sempre recebe o valor como sendo do tipo ***string***. Por isso o resultado apresentado está sendo uma concatenação e não uma soma.

Para resolver esse ponto, e termos como resultado a nossa soma, teremos que realizar um procedimento que é muito comum em diversas áreas e linguagens de programação, que é a **conversão de tipos**.

Em outras palavras, precisaremos converter os valores de **n1** e **n2** de ***string*** para ***number***, dessa forma do JS irá interpretar o sinal de **+** como uma adição.

Existem algumas maneiras para se converter uma **String** para um **Number**, a primeira delas será através dos comandos:

* **Number.parseInt(n)**
* **Number.parseFloat(n)**

Onde a primeira delas fará uma conversão para um número inteiro enquanto a segunda para um número real.

Nesses comandos, **‘parse’** significa parsear, ou converter.

É importante lembrar que o JS é *case sensitive*, ou seja, ele diferencia e leva em consideração letras maiúsculas e minúsculas, então nesses casos é de extrema importância que o N de Number e o *I* e *F* de Int e Float, respectivamente, estejam em maiúsculo para que a conversão funcione.

Dessa forma, ao aplicarmos os comandos acima teremos:

**var n1 = Number.parseInt(window.prompt('Digite um número:'))**

**var n2 = Number.parseInt(window.prompt('Digite um segundo**

**número:'))**

**var soma = n1 + n2**

**window.alert('Você digitou os números: ' + n1 + ' e ' + n2 +'. A soma entre eles é: ' + soma +', correto?')**

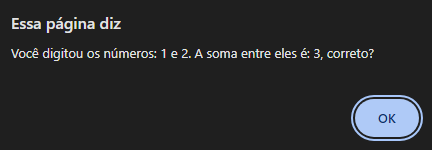
Dessa forma fazemos com que as nossas variáveis n1 e n2 se tornem numéricas inteiras.

Obs.: É importante lembrar que ao aplicar esse comando para conversão o conteúdo seguinte deverá ser colocado novamente entre parênteses.

E com isso teremos o seguinte resultado:







A soma é realizada da forma correta!

Porém uma grande limitação que esse método tem é a especificidade da declaração, pois estamos especificando que as variáveis além de números serão inteiras ou floats, então caso seja preenchido algum numera real em uma variável declarada inteira o sistema não irá calcular da forma correta. Para isso temos uma segunda maneira que unifica e resolve esse problema.

A segunda maneira é utilizando somente *Number*, ou outro tipo de dado que queremos, antes da declaração. Nesse caso estamos informando ao JS que aquela variável é numérica, independentemente de ser inteira ou float. Nesse caso estamos deixando o JS decidir dependendo do valor que o usuário preencha.

Essa maneira é um bom recurso para campos de preenchimento que não estão especificados se aceitam somente números inteiros ou não, sendo assim o navegador irá calcular considerando os dois tipos.

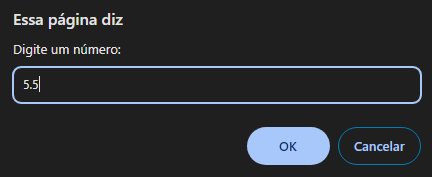
**var n1 = Number(window.prompt('Digite um número:'))**

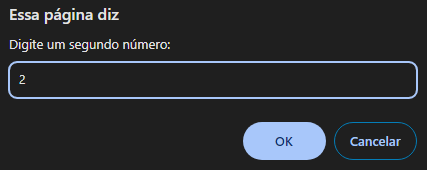
**var n2 = Number(window.prompt('Digite um segundo número:'))**

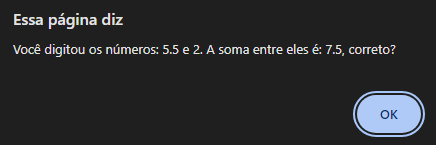
**var soma = n1 + n2**

**window.alert('Você digitou os números: ' + n1 + ' e ' + n2 +'. A soma entre eles é: ' + soma +', correto?')**

Então:







O JS automaticamente identificou que foram inseridos números inteiros e reais e ainda sim realizou a soma.

E se quisermos fazer ao contrário? Converter um ***número*** para ***string***.

Para esse caso temos também duas maneiras, a primeira delas e mais moderna é bem similar a essa última que aprendemos, onde inserimos o tipo **String(n**) a frente da declaração que queremos converter.

A segunda, que se trata de uma forma mais antiga, é a **(n).toString()**.

Aplicando-a ao nosso código anterior, teremos as seguintes declarações.

Método 1:

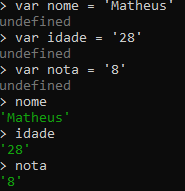
**window.alert('Você digitou os números: ' + n1 + ' e ' + n2 +'. A soma entre eles é: ' + String(soma) +', correto?')**

Método 2:

**window.alert('Você digitou os números: ' + n1 + ' e ' + n2 +'. A**

**soma entre eles é: ' + (soma).toString() +', correto?')**

Ainda no tópico de formatação de strings, o modelo e concatenação utilizando o + pode ser um anto quanto trabalhoso caso precisarmos declarar uma frase muito grande, conforme exemplo realizado no node abaixo.





Para isso utilizamos uma técnica de formatação mais pratica e moderna em JS o ***Template Strings***.

Para essa técnica, utilizamos a crase e não mais as aspas simples, pois a crase funciona como delimitador do *template strings* e utilizamos uma combinação de símbolos chamada *placeholder*: **${ }**

Dessa forma teremos o mesmo resultado anterior, porem com uma escrita mais pratica e dinâmica.



Da mesma forma conseguimos aplicar essa técnica ao nosso alerta do exercício acima:

**window.alert(`A soma entre ${n1} e ${n2} é ${soma}!`)**



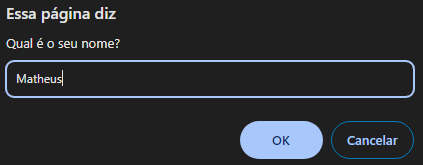
Obs.: Essa técnica apesar de ser menor e mais simples não quer dizer que seja melhor, as duas tem a mesma funcionalidade. Porem essa ultima deixa a linha de código mais limpa.

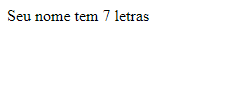
Outras técnicas que podemos usar para formatar uma ***string*** são:

* n**.length** 🡪 Nos informa quantos caracteres a *string* tem;

**var nome = window.prompt('Qual é o seu nome?')**

**document.write(`Seu nome tem ${nome.length} letras`)**





* n**.toUpperCase()** 🡪 Altera as letras da *string* para maiúsculo;

**var nome = window.prompt('Qual é o seu nome?')**

**document.write(`Seu nome em maiusculas é ${nome.toUpperCase()} <br>`)**



* n**.toLowerCase()** 🡪 Altera as letras da *string* para minúsculo.

**var nome = window.prompt('Qual é o seu nome?')**

**document.write(`Seu nome em minusculas é ${nome.toLowerCase()}`)**

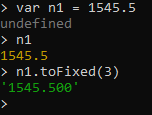


Obs.: n representa o identificador da variável.

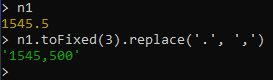
Obs.: Lembrando que PODEMOS usar tags de HTML dentro dos scripts de JS, conforme o uso do *<br>* acima para quebra de linhas.

Para números, podemos também aplicar as seguintes formatações:

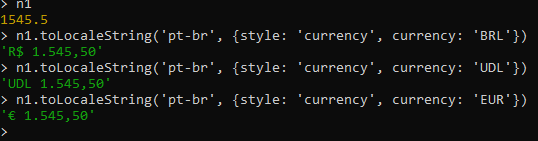
* n**.toFixed(n)** 🡪 para formatar o numero de casas decimais que queremos.



* n**.toFixed(n).replace()** 🡪 para alterar a exibição e formato de algum número, como por exemplo trocar o uso do ponto para o uso de virgula.



* n**.toLocaleString()** 🡪 para apresentar o formato do numero como por exemplo, um valor monetário local.



# Aula 7 – Operadores (Parte1)

Agora que aprendemos a diferença entre concatenação e adição utilizando o +, chegou a hora de aprender sobre os outros operadores e realizar contas em JS.

O JS possui várias famílias de operadores, como os **aritméticos**, os de **atribuição**, os **lógicos**, os **relacionais** e os **ternários**.

**7.1 – Operadores Aritméticos**

São eles os utilizados para se fazer cálculos.

* **Adição ( + )** 🡪 Realiza uma soma comum.

****

* **Subtração ( - )** 🡪 Realiza uma subtração comum.

****

* **Multiplicação ( \* )** 🡪 Realiza uma multiplicação comum.

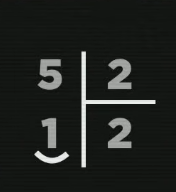
****

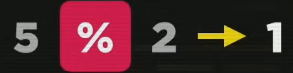
* **Divisão de números reais ( / )** 🡪 Seria essa a divisão comum, como conhecemos no dia a dia. Essa forma de divisão reconhece números reais, portando valores quebrados e com virgula.



* **Resto da Divisão de números inteiros ( % )** 🡪 A divisão de números inteiros, apesar do sinal de porcentagem, não calcula a porcentagem, mas sim o resto da divisão inteira.

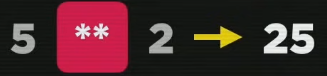
Esse calculoé realizado até que chegue o momento de se colocar uma virgula e é nesse momento que a conta acaba e o resto dessa divisão é o nosso resultado final, conforme exemplo:

****



Ou seja, 5 porcento 2 é o resto da divisão inteira de 5 por 2!

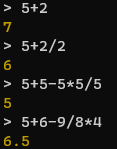
* Potencia ( \*\* ) 🡪 Significa “ao quadrado”



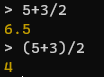
Logo, 5 \*\* 2 = 5² = 25.

Todos esses operadores são chamados de binários, pois precisam de dois operandos para se obter um resultado.

**IMPORTANTE**: Assim como na matemática que aprendemos na escola, em programação também existe a precedência de operadores, ou seja, se tivermos uma expressão com mais de um operador, como soma, subtração, multiplicação e divisão, a multiplicação e divisão são realizadas primeiro. Então devemos ter muito cuidado com isso.

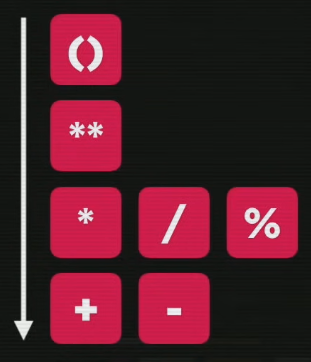


Mas se caso quisermos uma expressão onde de fato queremos realizar uma soma antes de uma divisão, por exemplo, para calcular uma media entre esse resultado, podemos utilizar, assim como na matemática os parênteses para informar aquilo que queremos a operação resolva primeiro.



Conforme exemplo acima, a primeira expressão segue a regra de precedência de operadores, realizando a operação de divisão primeiramente e após a soma. Já a segunda, com o uso dos parênteses, estamos informando que a soma deve ser realizada primeiro e na sequencia esse resultado, dividido por dois.

Dessa forma, em JS temos uma ordem de precedência que uma expressão segue para realizar as operações, que é a seguinte:



Primeiramente se calcula os parênteses, em seguida as potencias, após as multiplicações/divisões/resto da divisão e pôr fim a adição ou subtração.

Obs.: Note que a multiplicação, divisão e resto da divisão, estão no mesmo andar da hierarquia, ou seja, será calculado a operação que vier primeiro neste caso.

**7.2 – Operadores de Atribuição**

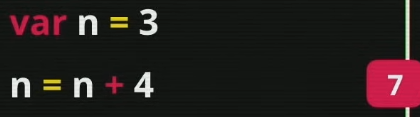
São utilizados para guardar dados das operações aritméticas. Conforme exemplos abaixo:



Onde estamos dando as variáveis de **a** à **f** valores de expressões aritméticas, que podemos utilizar posteriormente considerando somente seu valor final. Como por exemplo, var a = 8, var b = 3, etc.

Obs.: Destacados acima estão as operações que serão calculadas primeiros, seguindo a ordem de precedência que aprendemos anteriormente.

Outra função dos operadores de atribuição é a **auto-atribuição**, ou seja, a função de repor o valor guardado na variável após realizar uma nova operação.



No exemplo acima, estamos inicialmente atribuindo um valor simples a uma variável **n** e em posterior utilizamos para essa mesma variável uma nova operação. Nesse caso a variável que inicialmente recebia o valor de 3 passara a receber o novo valor de 7. É como se o valor guardado fosse atualizado.

Fazendo um paralelo ao exemplo do estacionamento da aula de variáveis, a lógica aqui seria a mesma, onde uma vaga só pode receber um novo carro se o anterior que estava lá sair.

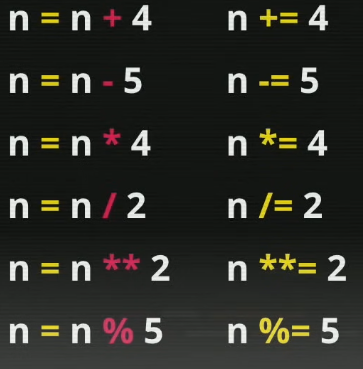
Também é possível escrever essa auto-atribuição de forma mais simplificada. Aqui neste exemplo estamos realizando apenas uma operação e troca de valor da variável, mas podem acontecer casos em que são realizadas diversas operações, sendo assim podemos escrever da seguinte forma:



Onde da mesma forma, será puxado o valor de **n** e somado a 4, resultando assim em um novo valor para a variável **n**.

Obs.: Essa forma simplificada só é válida quando a auto-atribuição é relativa a mesma variável, para variáveis diferentes ela não irá funcionar.

Outros exemplos de simplificação:

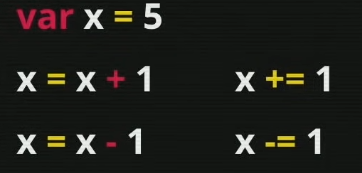


**7.3 – Operadores de Incremento**

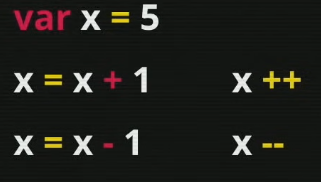
São operadores muito utilizados no cotidiano da programação, principalmente em uma programação mais avançada.

É muito comum adicionar o valor de *+/- 1* em alguma variável ao longo de alguns projetos, e para simplificar isso podemos utilizar a forma simplificada de operadores de auto-atribuição conforme aprendemos anteriormente, porém existe uma forma ainda mais simplificada e comum de se fazer isso, que são através dos operadores de incremento.

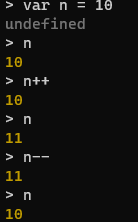
- Forma simplificada:



- Operador de Incremento:

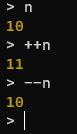


Em um exemplo pratico em NodeJS, criamos uma variável n que recebe inicialmente o valor de 10 e vamos incrementando-a através dos operadores de incremento, conforme abaixo.



Note que ao incrementar a variável com **n++** ela já não é exibida de imediato como 11, mas sim quando solicitamos novamente que a variável seja exibida que vemos o seu novo valor, a mesma coisa para o decremento **n--**. Isso porque os operadores **n++** e **n--** são valores de pós incremento ou decremento.

Porém podemos também realizar essa operação da seguinte forma:



Com **++n** e **--n**, a operação já é realizada de imediato, sem precisar atualizar a exibição da variável. Isso pois no formato acima, os operadores são de pré incremento ou decremento.

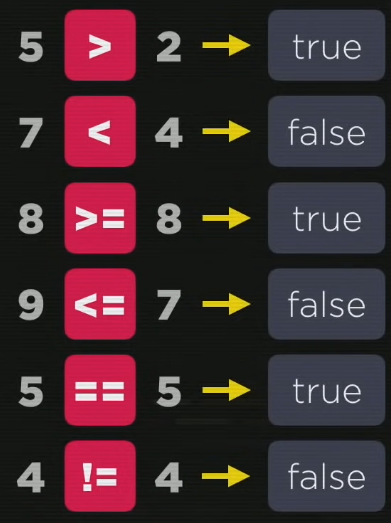
Isso não quer dizer que o formato n++ está errado, é apenas uma questão de ordem, pois existe o pré incremento/decremento e o pós incremento/decremento.

# Aula 8 – Operadores (Parte 2)

Dando continuidade ao aprendizado de operadores, nesta aula vamos continuar a aprender as utilizações e aplicações dos operadores relacionais, lógicos e ternários.

**8.1 – Operadores Relacionais**

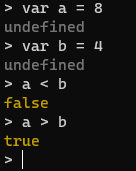
São operadores similares as demais linguagens de programação. Os operadores relacionais são utilizados para relacionar dois valores e o resultado de uma expressão relacional é um valor booleano, são eles: **maior**, **menor**, **maior ou igual**, **menor ou igual**, **igual** e **diferente**.



**IMPORTANTE**: Para toda expressão que tenha um operador relacional ligado a ela, o resultado dessa expressão será sempre um valor **booleano**, um valor do tipo verdadeiro ou falso.

Obs.: Lembrando que o símbolo de igual em JS (=) significa recebe, conforme aprendemos com variáveis. O valor de igualdade em JS é definido com o símbolo de igual duplo (==) conforme acima.

Importante ressaltar, que esses operadores de relação são comumente utilizados para realizar operações com variáveis, conforme abaixo.

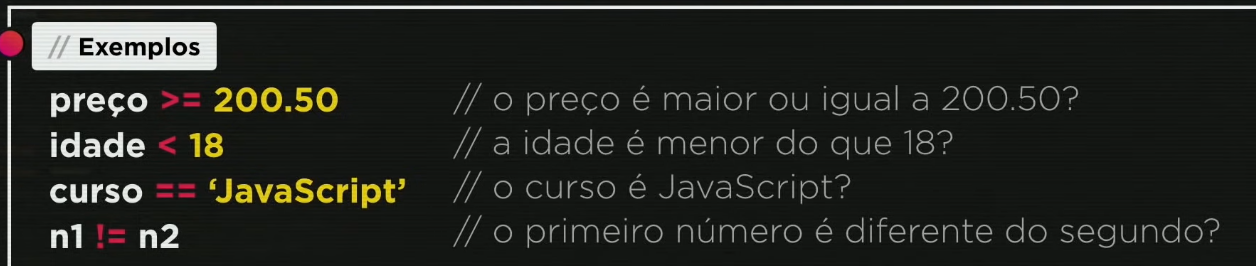


E até mesmo, mesclar operadores relacionais e operadores aritméticos na mesma expressão.



E quando essa situação ocorre, por ordem, primeiro são realizados os operadores aritméticos e na sequencia os relacionais para realizar a comparação.

Alguns outros exemplos que podemos utilizar, quando queremos aplicar alguma lógica de comparação em variáveis utilizando operadores relacionais e os que eles querem dizer.



8.1.1 – Operadores de Identidade

Uma outra característica dento dos operadores relacionais são os **operadores de identidade.**

Veja no exemplo abaixo:



Nele estamos comparando dois valores com o operador relacional de igualdade, porém na primeira expressão estamos comparando dois valores numéricos enquanto na segunda expressão o segundo valor é uma *string*.

Nesse caso os dois valores são verdadeiros porque o sinal de igualdade não testa o tipo. O que o JS faz nesses casos é analisar se o primeiro valor de 5 tem a mesma grandeza que o outro, sem levar em consideração o tipo.

Para isso, temos o operador de identidade, ou como também é chamado, **operador de identidade restrita**., que ao invés de dois sinais de iguais é representado por três sinais de igual (**===**).

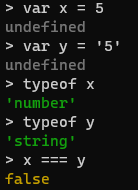
Esse operador valida se os valores são idênticos. Ou seja, de mesmo **valor** e mesmo **tipo**. Então utilizando o mesmo exemplo acima, teremos:



Na primeira expressão temos valores iguais porem tipos diferentes, logo é falso, não são idênticos.

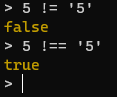
Na segunda expressão temos tanto valor quanto tipo idênticos, logo, verdadeiro.

Outros exemplos:



Da mesma forma, também temos o **desigual restrito**, onde a desigualdade (!=) não reconhece o tipo, apenas o valor. E a desigualdade restrita (!==) analisa valor e tipo.

Dessa forma:

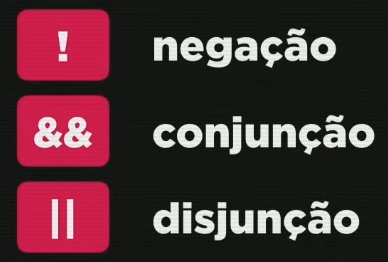


Na primeira expressão, estamos validando se o valor de 5 (number) é diferente do valore de 5 (string). O resultado é falso, porque esse operador apenas considera o valor, logo está validando que 5 é igual a 5 na expressão, portando falso.

Na segunda expressão, o operador está validando não só o valor, mas também o tipo, portando firma que é verdadeiro que 5 é diferente restrito de ‘5’.

**8.2 – Operadores Lógicos**

Em JS os operadores lógicos são 3: **negação**, **conjunção** e **disjunção**.

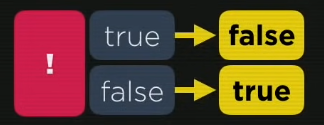


8.2.1 – Negação ( ! )

O operador de negação ou “não” é tratado como operador unário, ou seja, ele só tem um operando, isto é, após a exclamação ou se tem *true* ou *false*.

Sendo o valor da negação atendido (true), ou seja, não verdadeiro, o resultado é falso.

Sendo o valor da negação não atendido (false), ou seja, verdadeiro, o resultado é verdadeiro.

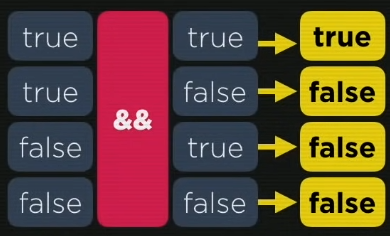


Um exemplo seria caso não se quisesse uma caneta vermelha, então, qualquer outro resultado que não seja uma caneta da cor vermelha satisfaz essa condição.

8.2.2 – Conjunção ( && )

O operador de conjunção, ou “e”, é um operador binário, onde se tem dois valores lógicos, um de cada lado, que trazem um resultado logico de uma expressão.

As condições desse operador só são satisfeitas quando os dois valores são verdadeiros (*true*).

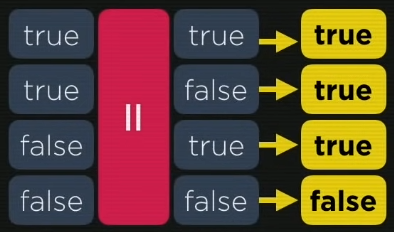


Um exemplo seria querer uma caneta azul **E** vermelha, ou seja, só é satisfeita a expressão quando de um lado (caneta vermelha) **e** de outro (caneta azul) são atendidos (*true* && *true*). Qualquer outro resultado não atende a condição, conforme acima.

8.2.3 – Disjunção ( || )

O operador de disjunção, conhecido como “ou”, também é um operador é binário, onde se tem dois valores lógicos, um de cada lado, que dependendo da posição deles e do resultado logico, trazem um resultado logico de uma expressão.

Na disjunção basta que um dos valores seja verdadeiro para o resultado ser verdadeiro.



Por exemplo, se nesse caso queremos uma caneta azul **OU** vermelha, quando qualquer um desses lados ou os dois é atendido, a condição já é satisfeita.

**IMPORTANTE**: Quando se tem operadores lógicos, relacionais e aritméticos numa mesma expressão, o resultado dela é calculado na seguinte ordem: primeiro o aritmético, segundo o relacional e por último o logico.



Expandindo a expressão acima para entendermos melhor o porque do resultado final ser *false*, temos o seguinte caminho:









Outros exemplos:

Ex01:

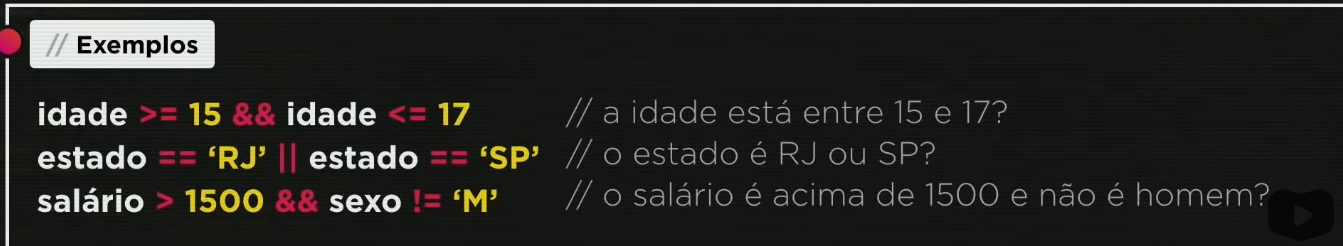








Ex02:



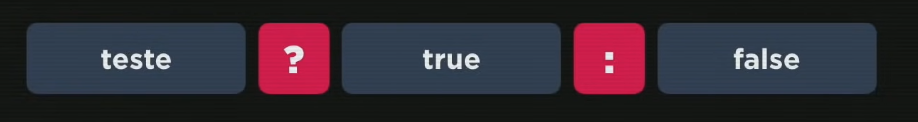
Vale lembrar, a ordem de precedência dos operadores com os que já aprendemos até aqui será:



8.3 – Operadores Ternários

O operador ternário é um operador que junta 3 operandos e é composto por dois símbolos que aparecem na mesma expressão: ? e :

Ele se chama ternário porque possui 3 partes, divididas em 3 blocos, conforme abaixo.

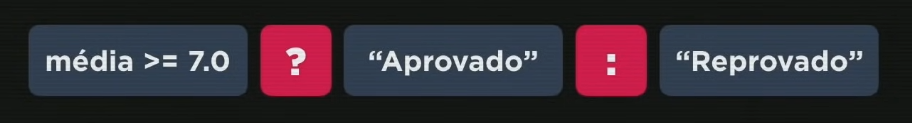


Sendo o primeiro deles o bloco do teste logico, um resultado que traz verdadeiro ou falso.

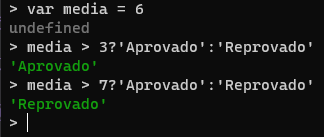
O Segundo é o bloco que diz o que vai acontecer se ele for verdadeiro.

E o terceiro bloco é o que diz o que acontece se for falso.

Ex01: Um exemplo prático é uma condição de média como teste logico, onde se o valor do teste logico atender a condição (true) a pessoa estará aprovada, caso não (false) estará reprovada.



Trazendo isso para o Node, temos o seguinte resultado:



Para uma pessoa que teve a media 6, caso a media fosse maior que 3 estaria aprovado, caso fosse maior que 7 estaria reprovado, conforme acima.

Ex02: Aplicando em uma expressão mais complexa.

Se atribuirmos um operador ternário a uma expressão com diversos outros operadores ele atribuirá o seu resultado a variável, por último. Conforme abaixo.



Nesse caso, já estamos definindo, através do operador ternário, que o resultado final para a *var x* será 5, caso o resultado do teste logico seja verdadeiro, ou 9 caso seja falso.

Assim como anteriormente, resolvemos primeiro os operadores aritméticos, então teremos:



Após, calculando o operador relacional, teremos:



Nesse caso o valor do bloco 2, verdadeiro, foi atendido, logo o resultado final da expressão e valor da variável *res* será 5.



Outros exemplos:



# Módulo C: Entendendo o DOM

Nesse módulo vamos aprender sobre o Document Object Model (DOM), que é a vertente web do JavaScript para o qual a linguagem foi criada.

É um poderoso conjunto de ferramentas que se encontra nos nossos navegadores. Um conjunto de objetos do JavaScript para o tratamento de componentes visuais, para se criar aplicações ainda mais bonitas.

* **C 🡪 Vamos aprender:**
* DOM (Conceito e aplicação);
* Árvore DOM;
* Estrutura DOM e manipulação;
* Eventos DOM.

# Aula 9 – Introdução ao DOM

Chegou a hora de começarmos a aprender sobre o DOM (Document Object Model), que é um poderoso conjunto de ferramentas que se encontra nos nossos navegadores. Um conjunto de objetos do JavaScript para o tratamento de componentes visuais internos de nosso site, para se criar aplicações ainda mais bonitas.

DOM (Document Object Model) é um facilitador que existe dentro do navegador, dentro do HTML5, para dar acesso aos elementos diretamente para o JavaScript.

Em outras palavras, é a vertente web do JavaScript para o qual a linguagem foi criada.

Obs.: O DOM não funciona em NodeJS, ele está presente apenas quando estamos rodando JS dentro do nosso navegador.

Mas antes de começarmos a dar os primeiros passos, precisamos instalar alguns recursos na nossa IDE, nesse caso o Visual Studio Code.

Primeiramente o Watch in Chrome 🡪 (DESCONTINUADO usar o Live Server) que servirá para abrir o nosso HTML com um micro servidor que será criado pela extensão.

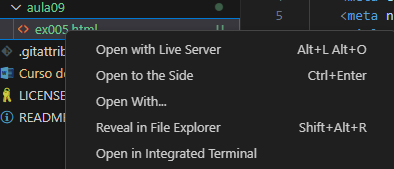
Esse micro servidor realizara em tempo real as atualizações que faremos no HTML. Sem precisar atualizar a página.

A segunda é o Node.js Exec que vai permitir utilizar a tecla F8 para rodar o nosso código JS pelo Node.

Agora que temos tudo pronto vamos dar inicio aos nossos primeiros passos em DOM.

Ex01: Apendendo a usar o DOM em JavaScript

Primeiramente, abrimos o nosso arquivo em *Live Server* (Extensão instalada) para que a página seja executada em um micro servidor, clicando com o botão direito no arquivo no VSCode e clicando na primeira opção *“Open with a Live Server”*.



Na sequência criamos um documento simples em HTML nesse arquivo, com um título, alguns parágrafos, uma *div* e uma área para os scripts que vamos criar.

**<body>**

**<h1>Iniciando estudos no DOM</h1>**

**<p>Aqui vai o resultado</p>**

**<p>Aprendendo a usar o <strong>DOM</strong> em**

**JavaScript</p>**

**<div>Clique em mim!</div>**

**<script>**

**</script>**

**</body>**

Adicionamos também um estilo local simples para a página na área de *head*.

**<style>**

**body {**

**background-color: rgba(0, 0, 0, 0.541);**

**color: whitesmoke;**

**font: normal 18pt Arial;**

**}**

**</style>**

E temos esse resultado.



**9.1 – Arvore DOM**

O DOM é como uma árvore genealógica, porém, de forma invertida. A arvore DOM começa da raiz, que dentro do navegador é a **Window**, ou seja, tudo dentro do JS está dentro de um objeto chamado **Window** (Janela do Navegador).

Dentro do Window temos 3 outros objetos:

**Location**: onde é informada a localização do seu site, qual a *url*, próxima página, página anterior, etc.

**Document**: é o documento que estrutura o seu site. O HTML que criamos.

**History**: guarda a navegação do seu site, de onde você veio ou para onde você vai.

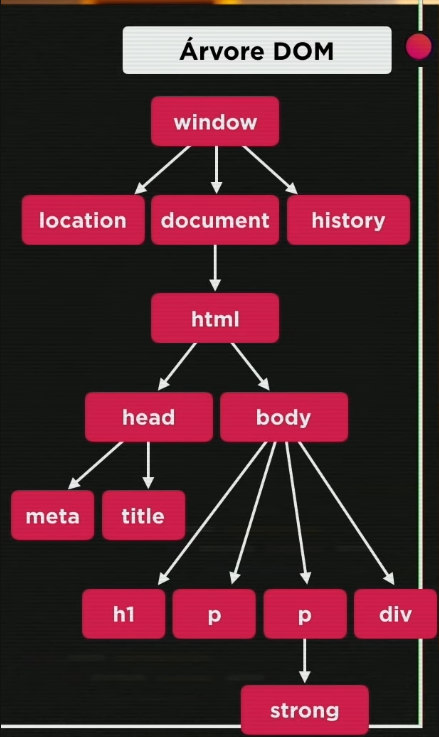
O elemento que antecede o *document* é o *window*, que nada mais é que a janela do navegador. Em sua estrutura, o *document* está no topo como objeto global e tem como elemento raiz a *tag* **html** e todas as outras descendem dela através das suas ramificações (*branchs*).

Uma árvore DOM, é uma árvore hierárquica, ou seja, teremos *parents* (pais) e *childs* (filhos).

A *tag* **html**, objeto pai, apresenta dois objetos filhos: o **head** e o **body** (o cabeçalho e o corpo). Os objetos que seguem nas ramificações de baixo são denominados como *child*, e os de cima, *parent*. A *tag* **head** é *parent* da *tag* **title**, e a **body** é *parent* das *tags* **a** e **h1**, e assim sucessivamente, de acordo com a hierarquia. Das *tags*, derivam os atributos, e destes, seus valores.

Levando como base o HTML que montamos acima, podemos definir, por exemplo, que *head* e *body* são filhos (*childs*) de HTML, assim como *h1*, *p* e *div* são filhos de *body*. Ou, que *p* é pai (*parent*) de *strong*, e assim sucessivamente.

Essa arvore compõe toda uma estrutura hierárquica dos elementos que compõem o nosso site.



E a pergunta que surge é: mas o DOM faz parte do HTML ou do JavaScript? Na verdade, de nenhum, ele é gerado pelo *browser*. Ao carregar a página, o navegador cria o documento, a interface, e o Javascript usa o DOM para se conectar ao HTML.

Para realizar a comunicação entre eles é necessário inserir a *tag* **script** no arquivo HTML, e como boa prática, ela deve estar antes do fechamento da *tag* **body** para que os *scripts* sejam carregados após o código base.

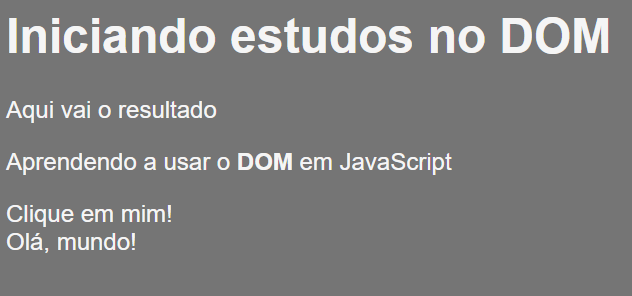
É possível realizar de duas formas: escrevendo o código em JavaScript dentro da própria *tag* script, ou inserindo o caminho relativo do arquivo externo. Também como boa prática, a segunda opção é a mais recomendada para a separação de responsabilidades e melhor manutenção do código.

Retomando ao exercício, podemos utilizar DOM diretamente de dentro do nosso JS. Como por exemplo:

**<script>**

**window.document.write('Olá, mundo!')**

**</script>**

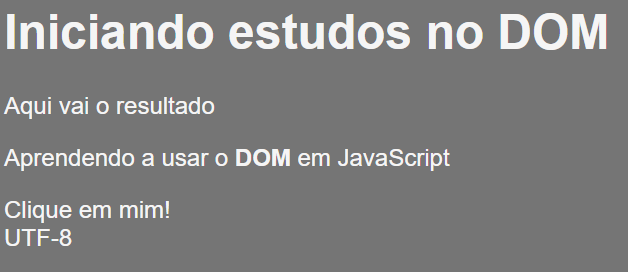


Onde pedimos para que seja escrito no documento “Olá, mundo!”

**<script>**

**window.document.write(window.document.charset)**

**</script>**

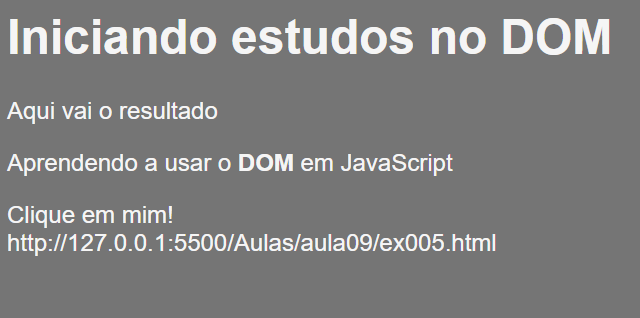


Onde pedimos para que seja escrito no documento qual o *charset* estamos usando.

**<script>**

**window.document.write(window.document.URL)**

**</script**>



Onde pedimos que seja escrito no documento a URL que estamos usando.

Note que em todos esses casos a estrutura hierárquica se mantem (**window.document.body.p.strong...**) e em cada um deles solicitamos ao componente **write** que exiba algum dos muitos objetos que existem na tela.

Dessa forma, conseguimos ter acesso a diversos componentes utilizando DOM diretamente do nosso JS e navegar dentro dessa arvore da maneira que acharmos melhor. E para isso existem varias maneiras de navegar entre os elementos.

Obs.: Elementos são todos os itens que estão presentes na nossa arvore acima.

Para navegar entre os elementos existem vários métodos, entre eles: **por marca (tag)**, **por ID**, **por nome**, **por classe** e **por seletor (CSS)**.

9.1.1 – Por Marca (tag)

Para esse método utilizamos o comando: ***getElementsByTagName( )***

Ele seleciona o elemento que queremos através do nome da tag que informarmos no comando.

Por exemplo:

**var p1 = window.document.getElementsByTagName('p')**

No exemplo acima, criamos uma variável chamada p1 e através dela guardamos o valor da tag ‘p’.

Porém, na maioria dos casos não usamos uma tag somente uma vez em nosso site, como por exemplo a tag de parágrafo, ou as *divs*, são tags que utilizamos constantemente em HTML. E se pedirmos para que o valor seja exibido ou faça parte de outra declaração, nesse caso, ele simplesmente não irá funcionar.

Para isso usamos esse método, nele precisamos sempre complementar no comando qual das tags em especifico queremos o valor. Para isso usamos, ao final, o valor de **[0]** caso quisermos o primeiro ‘p’, **[1]** caso quisermos o segundo ‘p ‘, **[2]** caso quisermos o terceiro ‘p’, e assim sucessivamente.

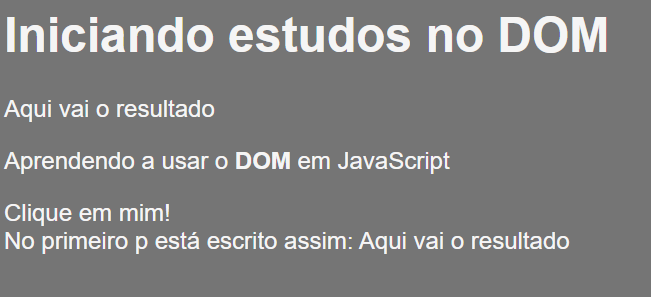
**var p1 = window.document.getElementsByTagName('p')[0]**

**window.document.write('No primeiro p está escrito assim: ' +**

**p1.innerText)**

Nesse exemplo estamos especificando que vamos selecionar o valor do primeiro ***p*** do documento (**[0]**) para nossa variável p1.

No comando abaixo estamos solicitando que escreva na tela o valor interno (**innerText**) da variável **p1.** Com isso temos:

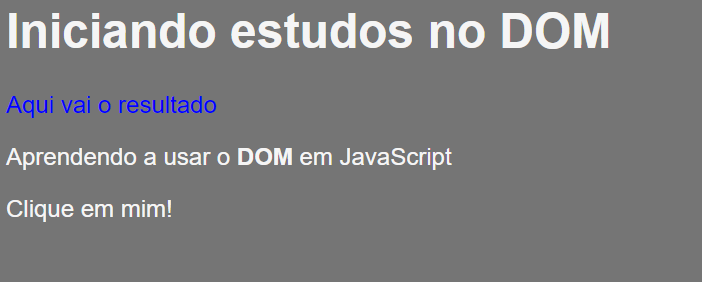


Mas nem sempre precisamos que um valor seja escrito ou exibido em tela, uma outra aplicação desse método pode ser simplesmente a alteração de um estilo dele, conforme abaixo:

**var p1 = window.document.getElementsByTagName('p')[0]**

**p1.style.color = 'blue'**

Novamente especificamos que o valor que queremos para nossa variável p1 é o valor do primeiro paragrafo do documento. E abaixo, estamos declarando que a cor desse paragrafo será alterada para azul. Dessa forma temos:



9.1.2 – Por ID

Para esse método utilizamos o comando: ***getElementById( )***

Assim como o anterior, utilizamos esse método para selecionar um elemento em especifico através do seu ID, seja uma div, um parágrafo, etc.

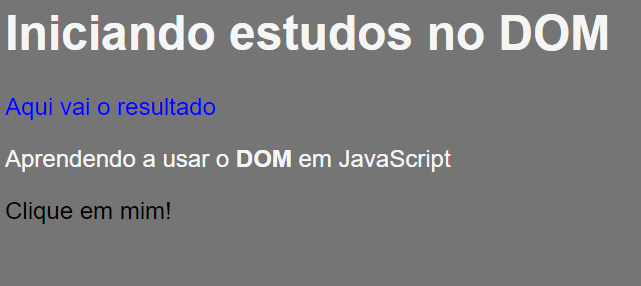
Obs.: Note que no primeiro metodo, por tag, o comando está no plural (getElementsByTagName) e por esse motivo precisamos especificar ao final de qual tag estamos buscando, uma vez que podemos ter mais de uma em alguns casos em nosso site. No caso do ID, como são únicos, não precisamos especificar.

Por exemplo:

**<div id="msg">Clique em mim!</div>**

**var d = window.document.getElementById('msg')**

**d.style.color = 'black'**

****

9.1.3 – Por Nome

Para esse método utilizamos o comando: ***getElementsByName( )***

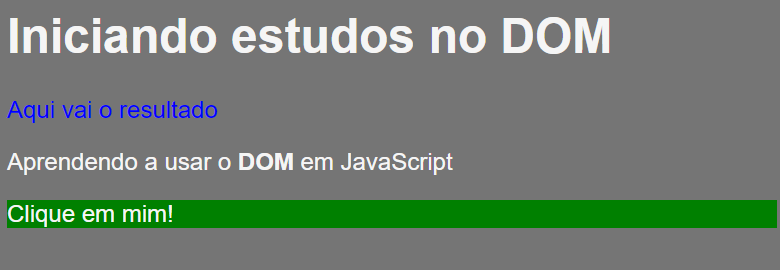
Utilizamos esse método caso tenhamos uma propriedade *Name*.

E assim como, por tag, podemos ter mais de uma dessa propriedade em nosso site, então, ao final, também precisamos especificar qual das propriedades *Name* estamos selecionando.

**<div name="msg">Clique em mim!</div>**

**var d = window.document.getElementsByName('msg')[0]**

**d.style.background = 'green'**



9.1.4 – Por Classe

Para esse método utilizamos o comando: ***getElementsByClassName( )***

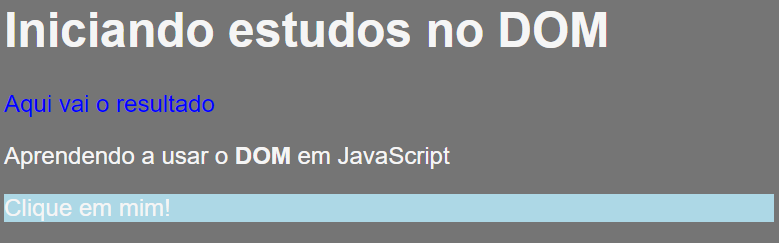
Utilizamos esse método para selecionar um elemento em especifico através da sua Classe.

E assim como, por tag e Name, podemos ter mais de uma classe em nossos elementos, então, ao final, também precisamos especificar qual das classes estamos selecionando.

**<div class="msg">Clique em mim!</div>**

**var d = window.document.getElementsByClassName('msg')[0]**

**d.style.background = 'lightblue'**



9.1.5 – Por Seletor (CSS)

Para esse método utilizamos o comando: ***querySelector( ), querySelectorAll( )***

Esse método é o mais recente de todos e um dos mais recomendáveis por muitos manuais.

Ele acaba se tornando mais pratico em muitos casos quando estamos realizando ajustes mais rápidos em alguns pontos do nosso site. Isso porque ele abrange as outras formas, podendo ser por ID, Class, etc., bastando usar a forma como esse seletor é declarado em CSS.

Por exemplo:

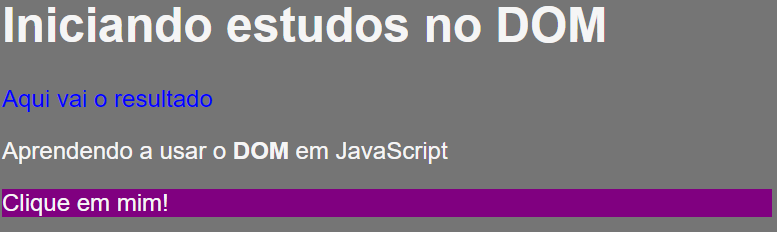
**<div id="msg">Clique em mim!</div>**

**var q = window.document.querySelector('div#msg')**

**q.style.background = 'purple'**

Nesse caso estamos utilizando o **querySelector** para selecionar a minha **div** de **id** **msg**, declarada como está representada em CSS.

Assim como, se tivéssemos uma classe, representaríamos como (**'div.msg'**).



Obs.: Utilizamos o ***querySelector( )***  quando for um elemento e o ***querySelectorAll( )*** quando for no plural.

# Aula 10 – Eventos DOM

Agora que conhecemos os conceitos de DOM e Arvore DOM, vamos aprender um pouco mais sobre seus eventos e pôr tudo isso em pratica.

Evento seria tudo aquilo que pode acontecer com um elemento.

Ex01: Aprendendo Eventos DOM

Primeiramente criamos um documento HTML simples, com apenas uma *div* no *body* e aplicamos a ela os seguintes estilos em CSS, para que fique com o conteúdo centralizado.

**<body>**

**<div id="area">**

**Interaja...**

**</div>**

**</body>**

**<style>**

**div#area {**

**background: rgba(0, 128, 0, 0.596);**

**color: white;**

**width: 200px;**

**height: 200px;**

**display: flex;**

**justify-content: center;**

**align-items: center;**

**font: normal 15pt Arial;**

**}**

**</style>**

Com isso, temos:



Agora, conceituando um pouco mais sobre Eventos DOM. Evento seria tudo aquilo que pode acontecer com um elemento, nesse caso, com a nossa *div*.

Entre muitos eventos um dos mais comuns são os eventos de mouse. Como por exemplo, um deles é mover o cursor do mouse até ele encostar, chegar dentro da *div*. Esse evento é o **mouseenter**.



Outro evento que podemos ter é quando continuamos movendo o mouse dentro da *div*, nesse caso temos o método **mousemove** sendo disparado diversas vezes enquanto estivermos mexendo o mouse dentro da *div*.



Outro evento é o **mousedown**, que é disparado no momento em que eu clico e seguro no botão do mouse dentro da *div*.

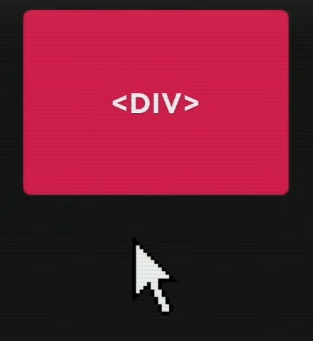


E no momento que eu soltar o botão é disparado outro evento, o **mouseup**.



Temos também, um dos eventos mais comuns, que é o ato de clicar normalmente, de forma rápida, em qualquer lugar dentro da *div*. Esse é o evento **click**.

Assim como o evento de entrada do mouse na *div* temos também um evento que é disparado na saída desse mouse da *div*, o **mouseout**.



Esses são somente 6 dos muitos eventos DOM que podemos ter em JS. Ao todo, são centenas de eventos nas mais diversas funcionalidades e elementos.

Para acesso a lista completa informada pelo próprio Mozilla, acessar o link abaixo.

**https://yari-demos.prod.mdn.mozit.cloud/pt-BR/docs/Web/Events**

Obs.: Os eventos podem ser configurados tanto na parte HTML quanto diretamente no JS.

Agora que entendemos o que é um evento e alguns dos que podemos fazer com eles, como no exemplo da nossa *div*, precisamos entender agora como dispara-los, como tratar um evento. E para isso vamos começar a entender o conceito de **Função** ou **Funcionalidade** em JavaScript.

**10.1 – Função em JavaScript**

Uma função em JS é um conjunto de código, ou conjunto de linhas, que serão executadas quando um determinado evento ocorrer.

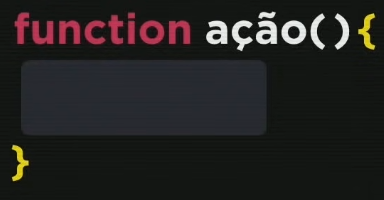
A quantidade de linhas que programamos na função é chamada de **bloco**.

Para executar essas linhas de código quando um evento ocorrer o primeiro passo é colocar essas linhas dentro de um bloco. Um bloco em JS é delimitado entre os sinais de chaves **{ }**.

Esse bloco, delimitado por chaves, precisa ser nomeado por uma **function**.

Para que o método possa funcionar, precisamos também dar um **nome** para essa função e geralmente os nomes que são dados as funções são ações que se podem fazer, funções de evento.

Dessa forma temos a seguinte estrutura para declarar uma função em JS:



Podemos também, colocar dentro dos parênteses alguns parâmetros. Sendo um ou vários.

Obs.: No JS existe o que chamamos de função anônima, que é uma função que não possui nome.

Sendo assim, continuando o exemplo acima como base, vamos praticar algumas funções em eventos DOM.

Ex01: Praticando funções (Configurações de Eventos em HTML)

Os eventos podem ser configurados tanto na parte HTML quanto diretamente no JS, fica a critério do desenvolvedor. Neste exemplo vamos configurar o evento de clicar, em HTML.

**<div id="area" onclick="clicar()">**

**Interaja...**

**</div>**

Na *div* que criamos anteriormente, adicionamos o evento **onclick**, onde demos o nome de **"clicar()"** que significa que ao clicar naquela área de evento (*div*) acontecerá alguma ação.

Ação essa que vamos declarar dentro da nossa função em JS abaixo.

**<script>**

**function clicar() {**

**var a = window.document.getElementById('area')**

**a.innerText = 'Cliquei!!!'**

**}**

**</script>**

Criamos uma função em JS, de nome **clicar()** (Nesse momento já estamos associando a ação a ser realizada nessa função ao evento que demos o mesmo nome relacionado a nossa *div*).

Sendo assim informamos uma variável, que chamamos de **a**, que recebe (Seguindo o conceito de arvore DOM) o caminho da nossa *div* de id **'area'** que, ao clicarmos, terá o seu texto interno(**innerText**) alterado para **'Cliquei!!!'**.

Dessa forma, teremos:

1. Div com o conteúdo inicial:



1. Div após clicarmos nela:



Outro evento que podemos exemplificar é o de passar o de entrar com o mouse. Nesse caso teríamos o evento configurado em HTML, onde queremos:

**<div id="area" onmouseenter="entrar()">**

**Interaja...**

**</div>**

E a função criada com a ação a ser realizada para esse evento:

**function entrar() {**

**var entrar = document.getElementById('area')**

**entrar.innerText = 'Entrei!!!'**

**}**

Nesse caso, ao entrar com o mouse na nossa *div* aparecerá a mensagem **'Entrei!!!'**.

Resultado:



Ao passar o mouse,



Nos exemplos acima, aplicamos variáveis diferentes para receber a nossa *div* de *id* ‘area’, porém em termos de praticidade usaremos a mesma variável para declarar ações na mesma *div*.

Porém se mantivermos a variável declarada em apenas uma função, por mais que tenhamos configurado em HTML vários eventos para aquele objeto, ela só funcionará para a função em que foi declarada. Conforme abaixo.

**function clicar() {**

**var a = window.document.getElementById('area')**

**a.innerText = 'Cliquei!!!'**

**a.style.background = 'blue'**

**}**

**function entrar() {**

**a.innerText = 'Entrei!!!'**

**a.style.background = 'red'**

**}**

Apenas funcionará a ação da primeira função, por mais que na segunda estejamos informando a mesma variável a, não a estamos declarando no bloco.

Dessa forma, existe um método para que todas as ações nas funções sejam realizadas sem precisar declarar a variável individualmente em cada uma, que é declarando a variável fora da função. Conforme abaixo:

**<div id="area" onclick="clicar()" onmouseenter="entrar()">**

**Interaja...**

**</div>**

**<script>**

**var a = window.document.getElementById('area')**

**function clicar() {**

**a.innerText = 'Cliquei!!!'**

**a.style.background = 'blue'**

**}**

**function entrar() {**

**a.innerText = 'Entrei!!!'**

**a.style.background = 'red'**

**}**

**</script>**

Dessa forma, teremos o mesmo resultado, pois a variável declarada fora da função responderá a todas ações que a chamarem.

Obs.: Essa forma não é a melhor forma de fazer pois pode se ter um problema de memora. Vamos entender isso mais a frente quando estudarmos sobre escopo.

Ex02: Praticando funções (Configurações de Eventos em JavaScript)

Além de disparar o evento em HTML, podemos também os disparar em JavaScript, utilizando ***listeners*** (**Event Listener**) que são ‘ouvidores’ que servem para listar os eventos dentro do JavaScript.

Então no lugar de fazer conforme abaixo, deixando o seu HTML poluído.

**<div id="area" onclick="clicar()" onmouseenter="entrar()" onmouseout="sair()">**

Podemos adicionar ao JS os seguintes comandos:

**var a = window.document.getElementById('area')**

**a.addEventListener('click', clicar)**

**a.addEventListener('mouseenter', entrar)**

**a.addEventListener('mouseout', sair)**

Dessa forma, mantemos o HTML mais limpo e o próprio JS vai tratar de relacionar os eventos com as funções criadas e o HTML.

Ex03: Detecção de erros

Agora que aprendemos esses conceitos, vamos também aprender um novo conceito que vai nos ajudar com situações, que a partir de agora, podem ocorrer bastante, a **detecção de erros**.

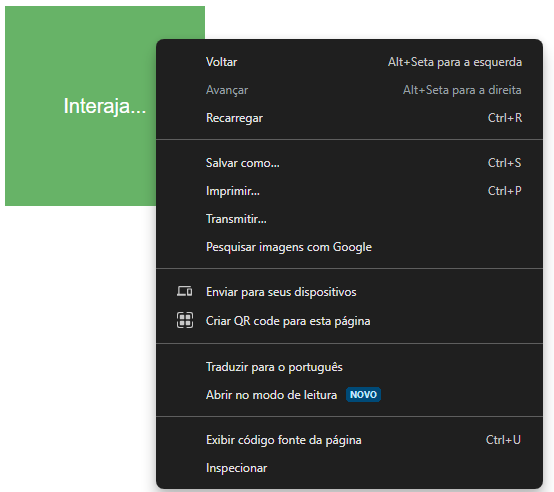
Com essas aplicações mais praticas e com necessidade de mais linhas de código para que uma interação funcione corretamente em JS, estamos também mais propícios a erros. E erros, dos mais simples, como uma letra maiúscula onde não devia ter, ou escrever rapidamente e acabar excluindo uma letra do comando, podem fazer com que simplesmente nada funcione.

Nesses casos o JS não nos dá o erro, não nos informa em tela que houve um erro quando ele acontece e nem onde ele está. Simplesmente a ação do comando não funciona.

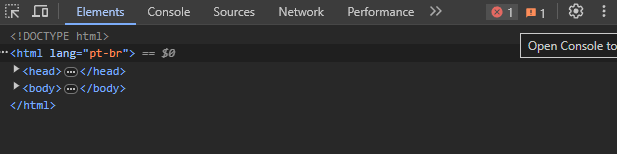
Vamos supor que em nossa variável acima, colocamos por acaso o W do window como maiúsculo. Isso fara com que todos os eventos e bloco de códigos não funcionem mais. E o JS e não irá nos informar.

**var a = Window.document.getElementById('area')**

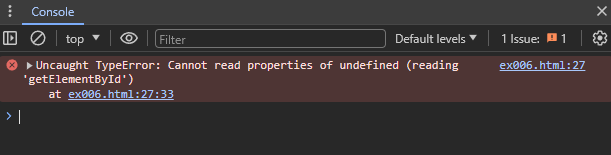
Nessa situação o meio mais comum de entender e se encontrar o erro é através do DevTools no próprio navegador.



Clicando na ultima opção, em ‘inspecionar’ o navegador abrira o DevTools e através dele, na aba principal, ele nos sinaliza se há erros ou não, além de onde eles estão, conforme abaixo.



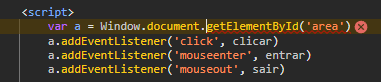
Ao clicar no ícone acima, será aberto o console com o erro informado:



No erro acima ele informa que “Não é possível ler a propriedade (lendo 'getElementById')”, encontrada na linha 27 (), que é justamente onde está declarado o *Window* com w maiúsculo.

E se clicarmos no link em azul informado no erro, o DevTools nos direcionará a exatamente onde aconteceu o erro dentro no nosso código na aba *Source*.

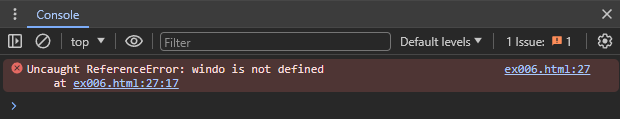
Ao clicar, nos é destacado a linha do código onde foi encontrado o erro.

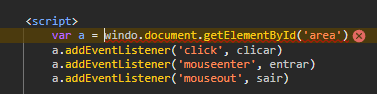


Outro exemplo, é quando escrevemos errado esquecendo alguma letra. Vamos supor uma nova situação onde foi escrito *windo*, sem o ultimo w, na mesma declaração de variável.

**var a = windo.document.getElementById('area')**

Nesse caso novamente o DevTools vai nos informar o erro, a linha que esta e a descrição, como por exemplo, neste caso.





É informado que “*windo is not defined*”, ou seja, ele não reconheceu a palavra ‘*windo’* na linha 27.

Ex04: Somando números (utilizando eventos, funções e variáveis)

Para concluir essa aula, vamos criar um novo exemplo utilizando eventos DOM, funções, variáveis, de forma a interagir mais com o usuário, buscando valores, etc.

Para esse exemplo, vamos criar um HTML para que seja realizado uma soma simples e que seja apresentada em tela.

**<h1>Somando Valores</h1>**

**<input type="number" id="txtn1" name="txtn1"> +**

**<input type="number" id="txtn2" name="txtn2">**

**<input id="botao" type="button" value="Somar" onclick="somar()">**

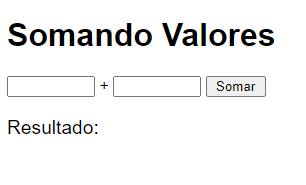
**<div id="res">Resultado: </div>**

Criamos 3 inputs, dois deles como *input:number* para que sejam preenchidos os valores em tela, um para cada valor da soma e um como *input:button*, para criar o botão que vai realizar o evento da soma quando clicarmos.

Para isso adicionamos a ele o evento **onclick** de nome **somar()**.

Criamos também uma *div* para que seja um bloco onde será exibido o resultado ao clicar.

Ao final, teremos o seguinte HTML.



Porém ainda não temos a funcionalidade da soma e para isso criamos o seguinte script em JS para que o evento funcione.

**<script>**

**function somar() {**

**var tn1 = window.document.getElementById('txtn1')**

**var tn2 = window.document.getElementById('txtn2')**

**var res = window.document.getElementById('res')**

**var n1 = Number(tn1.value)**

**var n2 = Number(tn2.value)**

**var soma = n1 + n2**

**res.innerHTML = `A soma entre ${n1} e ${n2} é igual**

**a <strong>${soma}</strong>. `**

**}**

**</script>**

Criamos a função **somar()** de mesmo identificador do evento configurado em HTML. E nela criamos 3 variáveis:

A variável **tn1**, receberá o primeiro valor do elemento **'txtn1'**, conforme *id* do primeiro *input*.

A variável **tn2**, receberá o primeiro valor do elemento **'txtn2'**, conforme *id* do segundo *input*.



A variável **res** recebe a nossa *div* criada para exibir o resultado de *id* **'res'**.

Porém, conforme aprendemos, para que a soma seja realizada corretamente, precisamos informar ao JS que os valores numéricos informados são do tipo *number*. Por mais que o input em HTML já seja desse tipo, se não convertermos em JS a soma não será realizada.

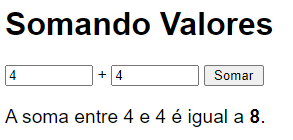
Para isso, criamos as variáveis **n1** e **n2**, que realizam a conversão das variáveis **tn1** e **tn2** para o tipo **Number**. Sendo assim, passamos a ter:



E com isso, conseguimos criar a nossa variável de **soma**, para realizar a operação entre os valores numéricos de **n1 + n2**.

Por fim, para que o resultado seja apresentado em tela, ao clicarmos no botão de soma, declaramos que **res.innerHTML** recebe a **soma** entre **n1** e **n2**, ou seja, a nossa *div* de resultado (**res**), ao clicarmos no botão “*Somar*”, irá exibir dentro de HTML o texto informado com o resultado da soma (variável **soma**).

Dessa forma, teremos como resultado:



Com um simples exercício, conseguimos praticar todos os conceitos aprendidos, como Eventos DOM, variáveis, conversão de tipos, arvore DOM e funcionalidades em um só lugar.

# Módulo D: Condições em JavaScript

A partir desse modulo D do nosso curso vamos trabalhar as condições em JavaScript.

Ele será composto de 5 aulas, incluindo exercícios, onde vamos tratar todos os tipos de condições que existem em JavaScript.

* **D** 🡪 **Vamos aprender:**
* Condições simples (estrutura *IF*)
* Condições compostas (estruturas *IF* e *ELSE*)
* Condições Aninhadas (*ELSEIF*)
* Condições Múltiplas (estrutura *Switch Case*)
* Exercícios propostos

# Aula 11 – Condições (Parte 1)

A partir dessa aula vamos falar das condições que temos em JavaScript.

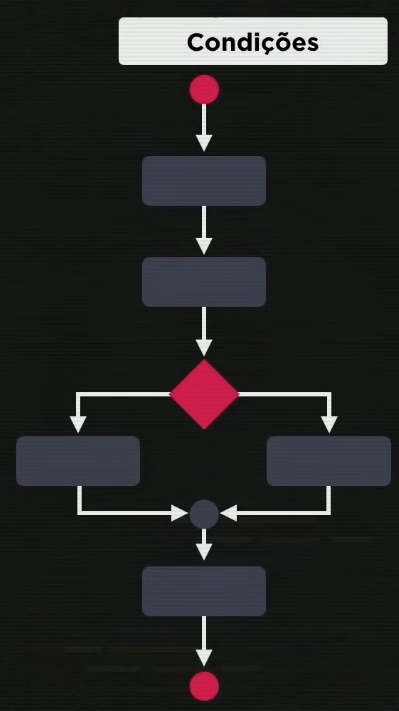
Mas antes de iniciarmos nesse assunto, precisamos falar sobre sequencias. Todos os códigos nos exemplos que realizamos até aqui foram códigos sequenciais, ou seja, as linhas de códigos possuem uma ordem e sequencia logica para que os resultados funcionem em tela.

Porém vão haver casos em que talvez vamos querer executar um comando e outra hora que não vamos querer executar aquele mesmo comando, mas ele estará declarado em nosso código, como faremos nesse caso para delimitar isso? É aí que entram as condições.

As condições funcionam como bifurcações ou desvios dentro do código e quebram essa linha totalmente sequencial dele. Com elas conseguimos, dada determinada condição, informar dois caminhos a serem seguidos a partir desse ponto, um ser for atendido uma ou a primeira condição e outro para uma segunda.

Condições também são conhecidas como **desvio condicional**.

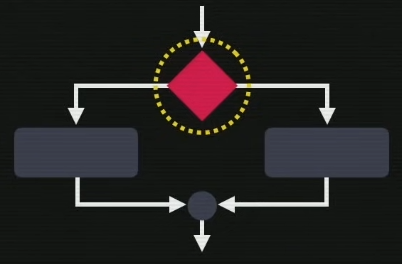
Abaixo a estrutura de um desvio condicional em um código inicialmente sequencial.



Essa bifurcação, que se inicia no losango acima, é justamente a nossa condição, que pode atender um dos dois caminhos para que siga na sequência do código mais à frente.

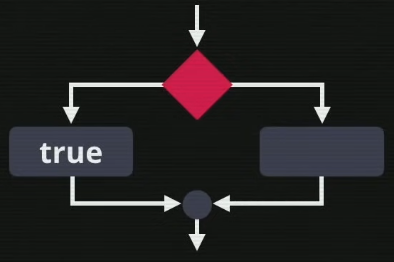
Mas como representamos isso no JavaScript?

Para o losango em JS vamos utilizar a estrutura **IF**(condição), que significa “se”. E essa condição nos trará duas possibilidades, que são representadas pelos dois blocos acima, um a esquerda (true) e outro a direita (false).

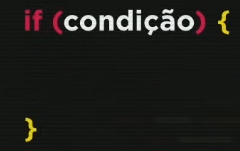


Esses blocos, como aprendemos anteriormente são linhas de código delimitadas por chaves **{ }**.

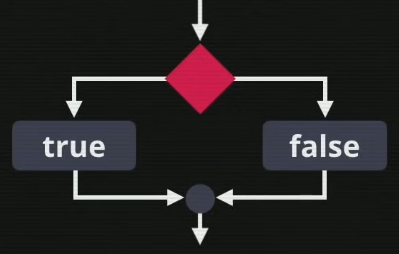
E para esse primeiro bloco, à esquerda, informamos a condição inicial a ser atendida se verdadeiro.



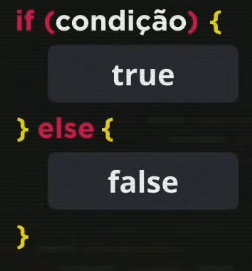
Com isso, formamos essa estrutura inicial de código abaixo:



Para o segundo bloco, à direita, utilizamos a palavra **ELSE** (se não...). Ou seja, se a primeira condição (*True*) não for atendida será atendida essa segunda (*False*).



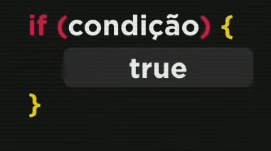
Com isso formamos a estrutura completa de condição em JavaScript, conforme abaixo:



Agora que aprendemos o conceito e a estrutura principal de uma condição, vamos aprender os tipos de condição que temos em JS.

**11.1 – Condição Simples**

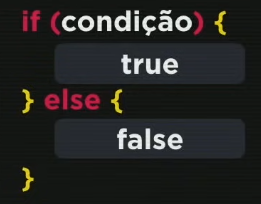
É uma estrutura de condição, como o nome já sugere, que é feita de forma mais simples, sendo representada apenas por um tipo de bloco (true). Conforme abaixo:



Nesse tipo de condição caso não seja atendida a primeira condição, ou seja, se a condição for falsa, nada irá acontecer. O fluxo do programa irá prosseguir normalmente.

**11.2 – Condição Composta**

Nesse tipo de condição trata-se da estrutura completa, conforme vimos acima. Teremos duas possibilidades a serem seguidas, representadas por dois tipos de bloco, *True* ou *False*.



Ou seja, se a primeira condição for verdadeira, acontecerá a ação, ou as ações, que estarão declaradas no bloco *true*. Se não, ou seja, ***else***, caso a condição seja *false*, irão acontecer as ações do bloco *false*.