Javascript é uma linguagem interpretada, ou seja, ela está sendo rodada em tempo real pelo navegador. Código disponibilizado instantaneamente (não precisa ser transformado em outra coisa antes de ser enviado para o browser ou console).  
  
Linguagens compiladas (C++) primeiro você coda primeiro, depois passa pelo compilador para que o computador consiga entender o código e aí depois vem o output do que aconteceu com aquelas instruções anteriormente passadas.

JS é uma linguagem multiparadigmas: pode trabalhar com orientação a objeto, programação funcional ou programação estruturada.

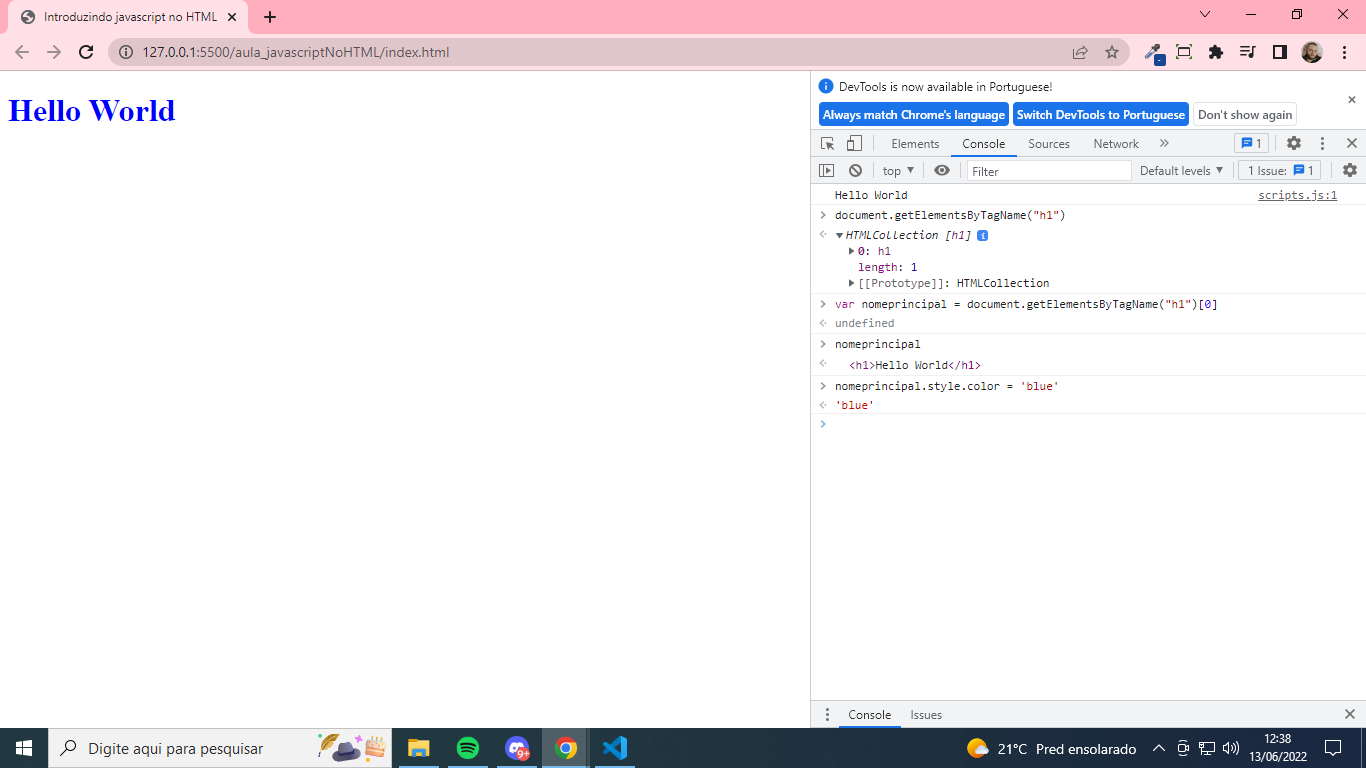
JS segue o padrão ECMAScript (conjunto de normas que vai dizer quais funcionalidades estão sendo lançadas naquela linguagem).

NOTAS   
- a declaração de “variáveis” constantes geralmente é feita com letras maiúsculas

ex: const PRECO = 2;

“variáveis” do tipo const não podem ser manipuladas, ou seja, seus valores são fixos…constantes!

funções - return: é utilizado quando o resultado da função precisa ser manipulado em alguma coisa (geralmente outra função)

Interagindo com elementos HTML e CSS por meio de códigos javascript no console do navegador. ↓  


Case Types (formas de nomear variáveis.

**Camel Case**: someAwesomeVar  
**Snake Case**: some\_awesome\_var  
**kebab Case**: some-awesome-var

**Pascal Case**: SomeAwesomeVar

**Upper Case Snake Case**: SOME\_AWESOME\_VAR (geralmente utilizado para variáveis do tipo CONST)

Atribuição de valores

Valores dinâmicos: Declarar com var e let

o tipo let tem escopo de bloco, ou seja, assim que ele sair do bloco a variável que foi declarada como let vai “parar de funcionar”

ex:

**var** a **=** **1**;

**var** b **=** **2**;

**if** (a **===** **1**) {

**var** a = **11**; //escopo global

**let** b = **22**; //escopo dentro do block if

**console**.**log**(a); //saída vai ser 11

**console**.**log**(b); //saída vai ser 22

}

**console**.**log**(a); //saída vai ser 11

**console**.**log**(b); //saída vai ser 2

Hoisting: atribuição de valor a uma variável antes de declarar ela(só funciona com o tipo var)

Redeclaração:

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

Javascript é uma linguagem de **tipagem dinâmica**, ou seja, antes de declarar um valor, você não especifica o **tipo** dele.

**Estrutura de dados** em *js*

Dois grandes grupos de dados: Primitivos e não-primitivos

**obs:tipos primitivos não possuem métodos dentro deles.**

**Primitivos**: números, strings(textos), boleanos, nulos(null) e indefinidos(undefined).

**Não-primitivos**: objetos e arrays(listas).

**alguns comandos**  
**typeof** - checar o tipo da variável

**Strings**

umaString**.concat**(outraString); - concatena a primeira string com a segunda

ex: let nome = “Bruno”

let sobrenome = “Henrique”

nome.concat(sobrenome);

saída: “BrunoHenrique”

umaString**.length** - retorna a quantidade de caracteres daquela string

**crase** facilita a utilização de strings com outros tipos de variávels

ex:

forma padrão

nomeCompleto = nome + “ ” + sobrenome;

saída: “Bruno Henrique”

com crase

nomeCompleto = **´${nome} ${sobrenome}´;**

saída: “Bruno Henrique”

algumaString**.split**(“”); - separa os argumentos da string. Retorna uma lista

algumaString**.includes**(“nome”); - verifica se a string possui a palavra nome. Retorna true ou false.

algumaString**.startsWith**(“B”); - verifica se o primeiro elemento da string é o caracter B. Retorna verdadeiro ou falso.

algumaString**.endsWith**(“?”); - verifica se o último elemento da string é o caracter ?. Retorna verdadeiro ou falso.

umNumero**.toString**(); - transforma uma variável do tipo número em uma string

Arrays

let umArray = []; - definindo um array vazio

umArray**.push**(3); - adicionando o número 3 no final do array

saída: 1 (saída foi 1 porque é o tamanho do array)

comando: umaArray;

saída: [3] (saída foi o elemento 3)

umArray.**unshift**(1); - adicionando o número 1 no começo do array

saída: [1, 3]

umArray**.pop**(); - retira o último elemento do array

saída: [1]

umArray**.shift**(); - retira o primeiro elemento do array

saída: []

**Coleções**

**Métodos e Maps VS Objeto**

Estrutura - map  
As estradas são no formato chave e valor. (possui uma série de argumentos)

Ex: **const** **myMap** **= new** **Map**( )

Métodos do map

Inserindo chaves e valores na const **myMap** ↓

**myMap.set**(**‘apple’**, **‘fruit’**);

saída: Map(1) {“apple” => “fruit”}

Checando valores ↓

**myMap.get**(**“apple”**);

saída: “fruit”

Deletando valores

**myMap.delete**(**“apple”**);

saída: true

Diferença entre maps e objetos   
\* Maps podem ter chaves de qualquer tipo, já os objetos possuem chaves nos formatos de strings.

\* Maps são mais fáceis de iterar

\* Os valores no map possuem o mesmo tipo

Coleção chaveada SET   
Estruturas que armazenam valores únicos (valores únicos que não se repetem nunca - principal diferença entre SET e array / para consultar o tamanho do set, usa-se a propriedade size no lugar de length / não possui diversas propriedades que um Array possui / Array é mais flexível )

Métodos do SET

**const** **mySet** **= new** **Set**( );

**mySet.add**(1); // adicionando no Set

**mySet.has**(1); // verificando se aquele set tem o valor 1

saída: true

**mySet**.**delete**(1); // deletando o valor 1 do Set

**Método MAP (arrays)**

A função de Map não modifica o array original

A função Map cria um novo array (o retorno dessa função será um novo array)

Realiza operações em ordem (do começo do array, index 0, até o último index)

Sintaxe

**array.map(callback, thisArg)**

callback: função que vai ser executada em cada elemento

dentro do callback temos acesso ao item, index e ao array original

ex: **callback(item, index, array)**

**thisArg(opcional):** valor de “this” dentro da função de callback

**Map vs forEach**

forEach também faz um operação em cada elemento do array

Diferença entre Map e forEach

**const** **array** = [**1**, **2**, **3**, **4**, **5**];

**array**.**map**((**item**) => **item** \* **2**);

saída: [2, 4, 6, 8, 10];

**const array** = [**1**, **2**, **3**, **4**, **5**];

**array**.**forEach**((**item**) => **item** \* **2**);

saída: undefined

Map retorna automaticamente uma nova array e o forEach não.  
Para utilizar o forEach deve-se considerar a criação de uma variável (um novo array) para guardar os valores

**Método filter**

Funciona como se o array tivesse sido colocado em um filtro e só passa os itens desse array que correspondem a determinada condição

* Não modifica o array original
* Cria um novo array
* Retorna um novo array apenas com os itens que “passaram” pelo filtro (condições)

**Sintaxe**

**array.filter(callback, thisArg)**

**ex:**

**const** **frutas** **=** [‘**maçã fuji**’, ‘**maçã verde**’, ‘**laranja**,’ ‘**abacaxi**’];

**frutas.filter**((**frutas**) **=>** **fruta.includes**(‘**maçã**’))

saída: [‘maçã fuji’, ‘maçã verde’];

**Método reduce**

Método que não retorna outro array / Executa uma função em todos os elementos do array retornando um valor único.

**Sintaxe**

**array.reduce(callbackFn, initialValue)**

Callback: função a ser executada a partir do acumulador

initialValue: valor sobre o qual o retorno final irá atuar(opcional)

**const callbackFn = function** (**accumulator, currentValue, index, array**)

{

**array.reduce**(**callbackFn, initialValue**)

**}**