JavaScript + Lógica de Programação

Created	@September 9, 2021 1:27 PM (GMT)
Created by	
Tags	Front End

▼ História e Características

→ Nasceu em 1992

→ Criada por Brendam Eich a serviço da Netscape (navegador)

→ Surgiu para dar mais dinamismo às Páginas HTML (HTML e CSS deixavam as pags estáticas)

→ Possuí muitos paradigmas (forma como a linguagem fornece os recursos p/ o desenvolver soluções):

Orientada a Objetos
Funcional
Orientada a Eventos

→ Possui tipagem dinâmica e fraca (infere o tipo de dado com base no conteúdo que a Variável recebe - sem tipagem → operações com diferentes tipos de dado, sem levar muito em conta o tipo de cada um).

→ O ECMAScript encontra-se atualmente na versão 8 (ES8)
→ Hoje não somente utilizada para front-end → NodeJs (interpretador capaz de ler códigos JS do lado do servidor)

▼ Comandos Básicos e Variáveis

Comandos Básicos

→ Comentários em arquivos JS: **//** (para única linha) **/**/** (para mais de uma linha)
→ Variáveis: Guardar dados → podem ser atualizadas sempre que necessário.
Var **a1 =** carro1 (a1 é o **identificador**)
(**=** RECEBE)

var → Global - Hoisting (efeito de elevação)
let → limita seu escopo no bloco
const → constante → não pode ser alterado por uma atribuição e não pode ser redeclarada

→ **Nome de cada variável → Identificador**

Regras de **IDENTIFICADORES**

- Podem começar com letra, $ ou _
- NÃO podem começar com números
- É possível utilizar letras e números
- É possível utilizar letras e símbolos
- Não podem conter espaços (utilizar _)
- Não podem ser palavras reservadas (que o JavaScript utiliza como comandos)

Dicas de como criar **IDENTIFICADORES**

- Maiúscula e Minúscula fazem diferença
  Usar padrão Camelcase (myBaby)
- Escolher nomes coerentes para as variáveis
- Evite se tornar um 'programados alfabeto' ou um 'programador contador'

→ **Acessar NODE**

(rodar JS fora do Navegador - Extensão do VSCode)
→ no Terminal: node -v → verifica a versão
node → entra
.exit → sair

→ No VSCode: Terminal (barra superior)
New Terminal (abre o terminal na barra inferior)
Digita node.
.exit (fecha o Node)
exit (fecha o terminal)
F8 - iniciar

▼ Tipos Primitivos

→ São a base de um código
→ Utilizamos para representar algum dado "bruto"
→ Com esses tipos que somos capazes de construir estruturas mais complexas
→ Pode ser alterado conforme a necessidade do algoritmo

Number (tudo o que envolve números)

String (texto - tudo o que estiver entre "" - ex "11")

Boolean (true e false)
Null (Objeto vazio, não possuí dados)
Undefined (Não foi definido)
Object (Mapeamento entre chaves e valores → Chave sempre será string
→ valores de qualquer type
→ var person { "name":"carla",
"idade":28}
Array
Function (Capacidade de ser chamado)

→ Quando defino a Variável, não defino type → muda conforme o atributo.

→ Como saber que type é a variável? no node = typeof nomedavariavel / ou console.log(typeOf nomedavariável)

▼ Manipulando os dados

→ Guardando:

window.**prompt(**'Qual seu nome'**)** → sempre vai retornar uma string
var **name** = window.prompt('Qual seu nome')
window**.alert**('É um prazer' **+ name**); → **+** é Concatenação

→ Conversão de Type:

Number**.parseInt(**n**)** → nº inteiros
Number**.parseFloat(**n**)** → nº reais
Number(n) → O próprio JS decide

Numero → string:
String(n)
n**.toSting()**

→ Formatando string:

Template String: (no lugar de concatenação → **${``}**)
.lenght() → quantos caracteres a string tem
.toUpperCase() → tudo em letras maiúsculas
.toLowerCase() → tudo em letras minúsculas

→ Formatando Números:
.toFixed(2**)** → para duas casas após o .
.toFixed(2**).replace ("." , ",")** → trocar → ponto por vírgula
.toLocaleString('pt-BR', {style:'currency', currency: 'BRL'})

→ escreve na tela → **document.write()**

▼ Operações Básicas

→ Operadores Aritméticos:

- + (somar)
- - (diminuir)
- * (multiplicar)
- / (dividir)
- % (resto/ módulo → 5 % 2 = 1)

- ** (potência → $5^2 = 25$)

CUIDADO com a PRECEDÊNCIA de operadores (quem será realizado primeiro): **( ) → ** → * / % → + -**

→ Operadores de Atribuição:

- Atribuição simples → var a = 5 + 3 → 8
- Auto Atribuição → var n = 3
  n = n + 4 → passa a valer 7
  ou **n+** = 4
- Incremento → x+ + → incremento pós incremento
  x- -
  + +x → pré incremento
  - -x

→ Operadores Relacionais:

- >
- <
- ≥ (> =)
- ≤ (< =)
- ==
- ! =
- Operadores relacionais de identidade:
  === (igualdade RESTRITA idênticos em valores e types)
  ! == (desigualdade restrita)

== → igual
= → recebe

→ Operadores Lógicos:

- ! → negação → não → !true = false
- && → conjunção → e → só satisfaz se as duas condições forem verdadeiras
- || → disjunção → ou → precisa de uma condição para satisfazer

CUIDADO com a PRECEDÊNCIA: **! → && → ||**

CUIDADO com a PRECEDÊNCIA GERAL operadores: aritméticos → relacionais → Lógicos

→ Operador Ternário (três partes): indicado usar somente quando são códigos pequenos - um comando ou uma linha

- teste **?** true **:** false (teste lógico ? o que vai acontecer sendo verdadeiro : o que vai acontecer sendo falso)

Ex: `var média = 3`

```
média > =7 ? "aprovado" : "reprovado"
"reprovado"
```

▼ Recursos Nativos do Browser

→ `console.log()` → imprimir dado no console (mais comum pra testar código)

→ `console.info()` → imprimir dado no console

→ `console.warn()` → imprimir mensagem de urgência no console - letra amarela

→ `console.error()` → imprimir mensagem de erro no console - letra vermelha

→ `alert()` → exibir uma caixa de mensagem na tela → botão de OK
→ `confirm()` → exibir uma caixa de mensagem de confirmação na tela → botões de CANCEL e OK

→ `prompt()` → exibir uma caixa de pergunta e espaço para digitar info/ dado.

`\n` → quebra de linha.

▼ Condicionais

→ Estrutura de Controle → desvio condicional

→ **Condição Simples (se - if)**: somente um bloco de condição → Se a condição for falsa, nada vai acontecer e continua o fluxo como antes : `if(condição) {`

`Bloco`

`}`

→ **Condição Composta (se - se não / if - else)**: O else não pode existir sem o if.

```
if(condição) {
Bloco positivo
} else {
  O que acontece se for falso
 }
```

→ **Condições Aninhadas (else if)**: Ninho - um dentro do outro →

```
 if(cond.1) {
Bloco 1
} else if (cond.2) {
Bloco2
 } else {
Bloco 3}
 }
```

→ **Condição Múltipla (Switch)**: Funciona com valores fixos (não só sim/não → outros valores fixos) → expressão, não condição. → Útil em situações pontuais e específicas (funciona com nºs inteiros e strings)

Dentro de cada bloco, **preciso** colocar o comando `break;`

```
Switch(expressão) {
  case(valor1): bloco1
    break;
  case(valor2): bloco
    break;
  case(valor3) : bloco
   break;
 default: bloco;
}
```

▼ Repetições em JavaScript

Estruturas utilizadas para repetir um bloco de código:
→ Repetições com teste no início (While - enquanto)
→ Repetições com teste no final (do While - faça enquanto)
→ Repetições com Controle (for - para)

→ **While:**

```
while(condição){
bloco
}
```

Vai acontecer enquanto a condição for verdadeira. Quando a condição for falsa, encerra a repetição (laço quebrado)

→ **do While:**

```
do {
bloco
} while(condição)
```

Faça isso aqui, enquanto essa condição for verdadeira.
Vai executar o bloco de código pelo menos uma vez, porque entra no bloco antes da verificação.

→ **For:**

```
for (inicio;teste;incremento){
bloco
}
```

ou

```
for ( variável; expressão; ação de controle){
bloco de código
}
```

Inicia com teste lógico, executa o bloco, vai fazer incremento e realiza novamente o teste lógico.
Quando o teste lógico ficar falso, encerra o loop.

→ **Utilizando o break:**

Encerrar uma estrutura de repetição a qualquer momento.

▼ Funções

São ações executadas assim que são chamadas ou em decorrência de algum evento. Pode receber parâmetros e retornar um resultado.

Funcionalidades - tarefas / rotinas - bloco de código para executar uma determinada ação.

Esse bloco é nomeado

Podemos chamar onde e quando precisamos

A declaração e a chamada de uma função seguem esse modelo:

```
function algumaTarefa(){
Bloco de código
}
```
→ declaração

```
algumaTarefa()
```
→ chamada

→ **Parâmetros das Funções:**

Receber dados ao serem chamadas - o que entra e o que você quer.

Uteis para quando queremos que a função execute uma tarefa, mas utilize algum dado que estamos enviando pra ela.

```
function parimpar(n)
```
→ parâmetro

```
{
if(n%2 ==0){
 return "par"
} else{
  return "impar"
}
let res = parimpar(11)
```
→ parâmetro

```
;
```
//impar

Parâmetros passam a ser variáveis dentro de funções. → Não são obrigatórios.

Parâmetros pré definidos/ Valor padrão → será o valor apresentado caso não seja informado um valor na chamada

```
function soma (n1 =0 , n2 = 0)
```
→ parâmetro pré definido -> opcional

```
{
 return n1 + n2
}
console.log (soma(2))
```
→ retorna com parâmetro definido, caso seja informado. Caso não seja informado parâmetro pré definido, retorna NAN(not a number). //2

```
let v = function (x) {
 return x+2
}
console.log(v(5))
```
//10

→ **Retorno das Funções:**

Retorna um valor para utilizar futuramente em um local

→ **Funções Anônimas:**

Sem nome - Armazenar dentro de variáveis.

```
let doubleSpeed = function( velocity ) {
return  velocity  * 2
}
let newVelocity = doubleSpeed(40)
console.log(newVelocity)
```

A função normal é lida antes de tudo no código(pode ser chamada antes no código), a função anônima somente quando chamada (segue a regra de uma chamada de variável)

→ **Arrow Functions: (⇒)**

Usar quando funções anônimas ou callback (uma dentro da outra → parâmetros de outras funções)

Variável Nome Função = parâmetro (se for um, não precisa de ()- se não tiver, deixar vazio ()) ⇒ Return (se for mais de uma linha, abrir bloco ⇒ { })

```
const soma = (num1, num2) ⇒ num1 + num2
console.log(soma(3,5))
```

```
const sayHello = name => `Hello ${ name }`
console.log(sayHello(`Carla`))
```

→ **High Order Functions HOF:**

Função de alta classe - recebem ou retornam outras funções
As funções que uma HOF recebem, geralmente são anônimas (podendo ser arrow ou não)
A função que é enviada como parâmetro de uma HOF é chamada de callback

▼ Scope

	VAR	LET	CONST
ESCOPO GLOBAL	V	X	X
ESCOPO DE FUNÇÃO	V	V	V
ESCOPO BLOQUEADO	X	V	V
PODE SER RESSIGNIFICADO?	V	V	X
PODE SER REDECLARADO?	V	X	X
PODE SER HOSPEDADO?	V	X	X
	Hosting → efeito de elevação	Limita seu escopo no bloco	Não pode ser alterado por uma atribuição Inicializar → guardar um dado Não pode ser redeclarada

▼ Arrays

Array é uma estrutura de dados, capaz de armazenar e organizar outros dados

Variáveis compostas → vários valores dentro de uma estrutura (não vai perder os valores anteriores para guardar novos valores) → reduzir código
Um array é um vetor/ variável composta que tem vários elementos, cada elemento é composto por seu valor e por uma chave de identificação

Dados são organizados em forma de lista (cada valor em uma determinada posição)

Pode armazenar internamente qualquer outro tipo (number/ string/ boolean/ outro array)

```
let num = [5,8,4]
```

valores = 5 8 4
elementos= 0 1 2

OU

`let num = new Array(5,8,4)` → perfeito para criar um array de texto ou um **array vazio**(se for array numérico, precisa tomar cuidado, porque se estiver com um item só (`new Array(5`) → vai criar um array com 5 posições `undefined` → vazio).

- Acrescentar valor:

  `num[3] = 6` → acrescenta 6 na posição 3 → `let num =[5,8,4,3]`

  `num.push(7)` → vai acrescentar o 7 no **fim** do array

  `num.length` → saber o tamanho da array (quantos elementos/posições tem)

  `num.sort()` → pega todos os elementos e coloca em ordem crescente

- Mostrar de forma bonitin no console:

  ```
  let num = [4,5,6,7,8]
  for (let pos=0; pos<num.length; pos++){
     console.log(num[pos])
  }
  ```

  OU (mesma funcionalidade simplificada):

  `for(  in){}` → para/ dentro

  ```
  for (let pos in num){
     console.log(num[pos])
  }
  ```
  → para cada posição dentro de um numero

  Resultado no console:

  1

  2

  3

  4

  5

- Buscar valores dentro de um array:

  `num.indexOf(7)` → 7 onde está o valor → retorna a Key(chave) onde o 7 está → se não encontrar o valor, retorna -1

▼ Funções básicas de **Arrays**

`.push(elemento)` → vai **acrescentar** novo elemento no **fim** do array

`.pop()` → **remove** o elemento que está na **última** posição do array → também retorna o elemento que foi removido (chamar dentro de uma variável) → ex: `let removedNumber = nomeArray.pop()`

`.shift()` → **remove** o elemento que está na **primeira** posição → também retorna o elemento

`.unshift(elemento)` → **adiciona** um elemento no **começo** da array

`.length` → retorna o tamanho da array

`.indexOf()` → índice de (posição em que está) → onde está o elemento.

▼ **Arrays Bidimensionais → Matriz**

Array que armazena outro array →

```
let spaceships = [ ["Elemental", 7] , ["Helmet", 13] ]
                                    0                1
```

Acessar informações do array dentro do array → `console.log(spaceships [0][1])` → // 7

['', colossus, helmet, Elemental]

▼ **Splice e Slice → manipulação no array**

- Splice:
  - Emendar ou costurar
  - Substituí o array original

  `array.splice(index[, deleteCount[, elemento1[,..[, elementoN]]])`

    - espera o primeiro parâmetro que é um índice do Array
    - O segundo é a quantidade de elementos que queremos remover a partir deste índice
    - Os próximos são os elementos que queremos adicionar no lugar

    - Apenas o primeiro parâmetro é obrigatório
- Slice:
    - Fatiar ou dividir
    - extrai uma parte da array sem alterar o array original

    `array.slice([begin[end]])`

        - Possuí um primeiro parâmetro que é o índice inicial
        - O segundo parâmetro é o índice final
        - Retorna todos os elementos entre o elemento de índice "begin" e o anterior ao elemento "end"

▼ Iteração em Arrays

- HOF → enviamos callbacks (funções que enviamos como parâmetros)
- O callback é chamado para cada elemento do array

```
array.funcaoDeIterar(function(elementoAtual, indice, array ){
<escopo da punção>
})
```

Espera um callback que receba como parâmetro o elemento atual, o índice e o array completo

- .forEach() → percorre o array e, para cada elemento, vai chamar o callback
- .map() → modifica os elementos do array, um por um, sem alterar o array original
- .filter() → filtrar os elementos da array → aguarda um retorno booleano do callback para saber se o elemento passou no filtro ou não → True -> insere no novo arrau / false -> não insere → retorna um novo array
- .find() → vai retornar o primeiro elemento que satisfazer a condição → retorna o elemento em si
- .findIndex() → retorna o indice de um filtro
- .join() → junta todos os elementos de um array em uma string e retorna esta string

    `const elements = ['Fire', 'Air', 'Water'`];

```
console.log(elements.join());
```
// expected output: "Fire,Air,Water"

```
console.log(elements.join(''))
```;
// expected output: "FireAirWater"

```
console.log(elements.join('-'));
```
// expected output: "Fire-Air-Water"