**Introdução ao Javascript**

Com sintaxe semelhante ao Java, Javascript domina a web permitindo que sites carreguem conteúdos mais dinâmicos nas suas páginas. Através do NodeJs, Javascript se tornou uma linguagem tanto para Front-end quanto para Back-end.

JS é de tipagem dinâmica e fraca, ou seja, dinâmica por suas variáveis não serem definidas de forma explicita no código e não conhecidas/checadas em tempo de execução/compilação. E de tipagem fraca por não ter o tipo da variável bem definido e a própria linguagem poder alterar o seu tipo de dado, sem a intervenção direta do programador.

Tipagem Fraca exemplo em JS: const numberOne = "5"

const numberTwo = 5

console.log(numberOne + numberTwo)

// 55

Tipagem Estática – o tipo de variável deve ser declarado antes de se atribuir valores a ela e seu tipo não pode ser alterado.

Exemplo em C: int numero = 5

Tipagem Dinâmica – declara-se a variável e não o seu tipo. Não há a necessidade de declarar que é uma variável do tipo String, por exemplo. Pode-se atribuir outro tipo de dado a variável também sem necessidade de alterar antes o seu tipo.

Exemplo em JS: var numero = “cinco”

var numero = 5

Tipagem Forte – os tipos das variáveis são bem definidos e só podem ser alterados com intervenção do programador.

Exemplo em Python: numero1 = “5”

numero2 = 5

print( int(numero1) + numero2)

#10

Por ser de tipagem dinâmica e fraca, JS possui uma linguagem simples e flexível, o que compromete um pouco sua clareza e manutenção de código.

Em JS funções são valores, portanto ela se torna uma linguagem própria para a programação funcional. Permite também Orientação a Objetos(OO).

Atualmente por rodar em navegadores, servidores(Node) e mobile(React Native) o JS faz parte do único stack (com adesão) que possui a capacidade de fazer front-end, back-end e mobile com apenas uma linguagem.

**Input**

let input = prompt("Digite um número");

alert("O número é: "+input);

**Variáveis**

Linguagem estática – a variável é de um tipo exclusivo de informação, não podendo ser alterada para outro tipo, por exemplo uma String não pode se tornar do tipo Inteiro, etc.

Linguagem dinâmica – a memória só aloca a variável se ela possuir um valor, e seu tipo pode ser alterado ao longo do código.

Tipos de variáveis em JS

Primitivos:

undefined -> Não há tipo nem valor

boolean -> verdadeiro ou falso

string -> textos

number -> números

Complexos:

function -> funções

object -> objetos

Em JS variáveis podem armazenar funções e objetos. Para uma padronização iremos utilizar daqui pra frente na declaração de variáveis *let* e na declaração de constantes *const*. Para conhecimento *null* é diferente de *undefined,* onde *undefined* é uma variável indefinida/não declarada e null ao ser atribuído declara a variável com valor vazio.

HOISTING

Ocorre quando o código JS é compilado, todas as variáveis declaradas são movidas para o topo do escopo local (se declaradas como função) e para o escopo global (se declaradas fora de uma função). Nesse processo declarações de funções e classes também são movidas para o topo ficando acima das declarações de variáveis.

Em termos de hoisting temos diferenças significativas entre VAR, LET e CONST:

VAR – pode ser declarado sem ser inicializado e está disponível fora de seu escopo

LET – pode ser declarado sem ser inicializado, porém não está disponível fora de seu escopo

CONST – não pode ser declarada sem ser inicializada e não está disponível fora de seu escopo

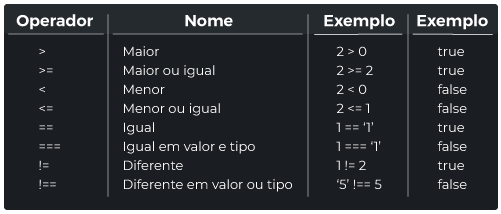
Ao utilizarmos *VAR* essas variáveis declaradas através dele não respeitarão seu escopo, ficando disponível fora dele, ou seja, uma variável declarada dessa forma, dentro de uma função, também estará disponível fora dessa função. Utilizando *LET* isso não ocorre, ou seja, o escopo é respeitado, tornando essa a principal diferença entre eles. Já o *CONST* impede que o valor seja alterado após ser iniciado, ao tentar muda-lo um erro irá aparecer.

Condicionais

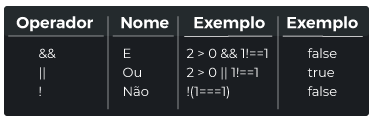
Operadores Lógicos

O código é executado de diferentes formas dependendo das condicionais inseridas dentro dele, que serão interpretadas como verdadeiras ou falsas dependendo de sua interpretação. É aí que surge o tipo de dado BOOLEANO que possui o valor *true* ou *false*.

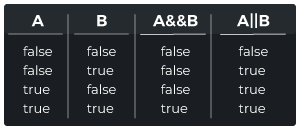
Portanto CONDIÇÃO é uma operação lógica cujo resultado é um valor booleano. Em JS temos os seguintes operadores de comparação:



Os operadores lógicos em JS são:



Os operadores de comparação retornam um booleano dependendo do resultado de comparação. Já operadores lógicos fazem operações sobre valores booleanos. Por exemplo && só retorna TRUE se ambas as condições forem verdadeiras. Já || retorna verdadeiro se uma das condições for verdadeira. Diante disse temos a seguinte tabela lógica.



If/Else

if(condição){ if(condição){

} }

else{ Ou else if(condição){

} }

else{

}

Operador Tenário (if/else)

É uma espécie de simplificação do if/else

Por exemplo: Há um cliente que com 100 pontos é um cliente premium, abaixo disso é um cliente comum.

let pontos = 200;

let tipo = pontos > 100 ? ‘premium’ : ‘comum’;

console.log(tipo);

Switch

Estrutura condicional que recebe um valor e executa o código conforme os casos que a variável receber.

let hoje = new Date().getDay();

let dia;

switch (hoje){

case 0:

dia = “Domingo”;

break;

case 1:

dia = “Segunda”;

break;

case 2:

dia = “Terça”;

break;

case 3:

dia = “Quarta”;

break;

case 4:

dia = “Quinta”;

break;

case 5:

dia = “Sexta”;

break;

case 6:

dia = “Sábado”;

break;

}

Truthy e Falsy

São casos que não há necessidade de testagem com valores específicos, eles já nos retornaram um True ou False. São exemplos de falsy variáveis que retornam FALSE são as que contém os seguintes valores:

* 0 (ZERO)
* -0 (menos Zero)
* '' (aspas simples vazias) ou "" (aspas duplas vazias) (string vazia)
* null
* undefined
* NaN (not a number)

Todos os demais retornam TRUE. São exemplos de truthy:

* []
* {}
* function(){}

Esses conceitos são utilizados para verificar se as variáveis são vazias, se possuem valores UNDEFINED, NULL, NaN, etc.

Laços de Repetição

Aqui temos os chamados Laços ou *loops,* que são estruturas utilizadas dentro da programação para eventos que poderão ocorrer mais de uma vez na execução/compilação do código.

While

Estrutura que executa o código enquanto a condição for verdadeira.

While (condição) {

Código a ser executado

}

Do-While

Há casos em que é necessário executar o *while* pelo menos uma vez. Para isso temos a estrutura *do-while*, onde primeiro se executa o código e depois é verificado a condição para continuar executando ou não. O código então será executado no mínimo uma vez. Então o WHILE verifica primeiro e executa depois, já o DO-WHILE executa primeiro e verifica depois.

Let i = 0;

Do {

Console.log(‘digitando!’);

I++;

}while (i < 10)

For

É uma estrutura de repetição que possui um número específico de vezes a serem executadas.

For ( let i = 0; i < 10; i++){

Console.log ( i + 1 );

}

Vetores (Arrays)

Sequência ordenada de valores:

let vetor = [1,2,3,4,5,6,7,8,9,10];

ou

let vetor = [ ];

vetor[0] = ‘teste’;

vetor[1] = ‘teste2’;

ou

vetor[‘indice1’] = ‘teste’;

vetor[‘indice2’] = ‘teste2’;

Vetores possuem a característica de serem heterogêneos, ou seja, comportam tipos de valores diferentes.

Let vetor = [1,2,3,’a’,’b’,’c’,true,false];

Para boas práticas construa vetores de tipos de dados únicos, ou seja, apenas de inteiros, apenas de Strings, apenas de booleanos, etc;

Percorrendo Vetores (Arrays)

Utilizando a estrutura do *for* é possível percorrer todo o vetor.

let vetor = [1,2,3,4,5,6,7,8,9,10];

for(let i = 0; i < vetor.length; i++){

console.log(vetor[i]);

}

Ou

FOR-OF

const cores = [‘vermelho1, ‘verde’,’azul’,’amarelo’,’roxo’,’laranja’];

for(let cor of cores){

console.log(cor);

}

Percorre os elementos do array e retorna o valor de cada elemento.

Ou

FOR-IN

Const pessoa = {

Nome = ‘Humberto’;

Idade = 25;

}:

For (let chave in pessoa){

Console.log(chave);

}

Percorre os elementos do array e retorna o índice de cada elemento.

Matrizes

Basicamente é um vetor de vetores (vetores multidimensionais).

const matriz = {

[1,2,3]

[4,5,6]

[7,8,9]

}

for (let linha of matriz){

for (let dado of linha){

console.log(dado)

}

}

Vetores Dinâmicos

Em JS vetores possuem em sua estrutura dados dinâmicos, ou seja, de n tamanhos.

**Para adicionar um novo elemento ao vetor:**

Let novo\_elemento = 60;

Vetor.push(novo\_elemento);

**Para remover elemento do final:**

Let vetor = [10,20,30,40,50];

Vetor.pop();

**Para adicionar ao começo:**

Let novo\_elemento = 0;

Vetor.unshift(novo\_elemento);

**Para remover do começo:**

Vetor.shift( );

Funções

Definimos funções em duas interpretações. Primeiro que é um trecho de código que nomeamos e que é executado a cada vez que seu nome é chamado.

// Definindo a função

function geradorNumerico(comeco, fim){

for(let i = comeco; i <= fim; i++){

console.log(i);

}

}

// Chamando a função

geradorNumerico(10,20);

Também interpretamos funções como matemáticas, onde recebem valores e devolvem outros valores.

function soma(a,b){

return a+b;

}

const resultado = soma(2,3);

São utilizadas as seguintes formas para se declarar funções

// Usando function

function soma1(a,b) { return a+b; }

// Atribuindo uma função anônima em constantes

const soma2 = function(a,b){ return a+b; };

// Atribuindo uma função de "flecha" em constantes

const soma3 = (a,b) => a+b;

A primeira foi declarada da forma tradicional. A segunda de forma anônima, onde não há atribuição de nome para a função, sendo o seu nome o nome da variável que a está recebendo.

No terceiro caso há também uma função anônima, porém a sua atribuição foi declarada como função de flecha, onde a sua atribuição é realizada utilizando os símbolos =>. Exemplos de função de flecha:

const hello1 = () => "Hello World!";

const hello2 = (name) => "Hello " + name;

const hello3 = (name) => {return "Hello " + name;};

Functions vs Arrow Functions

* Arrow Functions são sempre funções anônimas;
* Arrow Functions prometem entregar o mesmo resultado de código, com uma menor quantidade de linhas;
* Não é necessário utilizar a palavra function, a palavra return também não é necessária, ela está implícita, também não necessidade de chaves se for apenas uma instrução de código;

const sum = (a, b) => a + b;

const getRandomNumber = ( ) => Math.random( );

* Elas também redefinem como o *this* é tratado dentro da função. Elas não possuem o seu próprio *this;*

Funções de Alta Ordem (High Order Function)

Em JS funções são tratados como cidadãos de primeira classe, sendo assim podem ser tratados como valores e salvos como variáveis. Por terem esses privilégios outras funções podem ser passas como parâmetros para uma função.

O conceito de Função de Alta Ordem é uma função que recebe ou retorna uma função.

const somar = (a, b) => a + b;

const subtrair = (a, b) => a - b;

const aplicarOperacao = (a, b, operacao) => operacao(a,b);

const resultado1 = aplicarOperacao(1,2,somar);

const resultado2 = aplicarOperacao(1,2,subtrair);

No código acima declaramos três funções, *somar*, *subtrair* e *aplicarOperacao*. As duas primeiras recebem dois parâmetros e devolvem um valor. A *aplicarOperacao* recebe três parâmetros, sendo o terceiro uma função a ser aplicada passando os dois primeiros parâmetros. Nas linhas 5 e 6 são passados, respectivamente, as funções *somar* e *subtrair*.

**Retornando Funções**

Nós podemos retornar funções como uma forma de construir abstrações mais complexas, onde uma função é um geradora de funções. No exemplo a seguir, temos a função *somarX*, que recebe *x* de parâmetro e retorna uma função que recebe *y* e retorna *x+y*. Note que ao passar 1 para *somarX*, criamos uma função que soma 1. E assim sucessivamente.

const somarX = (x) => (y) => x + y;

const somar2 = somarX(2);

const somar3 = somarX(3);

console.log(somar2(5));

A função somarX está recebendo um parâmetro (x) que também é uma função que retorna => (y), que é uma função que retorna x+y. Então se passarmos 2 como parâmetro para X o retorno será y+2. Portanto ao passarmos 5 como parâmetro ao somar2, temos 5+2, o retorno é 7. O mesmo acontece ao somar3 que recebendo 5 como parâmetro retorna 8.

Map

É primeiro caso de função de alta ordem a ser estudado. Essa função é utilizada para transformar vetores. Ele é uma função específica de vetor para manipular seus itens. Pode receber até três parâmetros. Caso receba um parâmetro, esse parâmetro representa o valor do elemento do vetor. Caso receba 2, representa o valor do elemento e sua posição. Caso sejam os 3, representa o valor, a posição e o vetor inteiro. Por exemplo caso seja preciso um segundo vetor que é dobro do primeiro fazemos:

const vetor = [1,2,3,4,5,6];

const dobro = (item)=>2\*item;

const vetorDobrado = vetor.map(dobro);

O map não altera o vetor original, portanto colocamos o seu resultado em um novo vetor. Podemos também passar dois parâmetros para funções mapa, sendo a primeira o elemento, a segunda a posição no vetor e o terceiro o próprio vetor.

// Função que eleva ao quadrado

const aoQuadrado = (item, indice, vetor) => vetor[indice]\*item;

Também é possível passar a função dentro do próprio map.

const vetor = [1,2,3,4,5,6];

const vetorTransformado = vetor.map((x)=>x+1);

O map exige que ao menos um parâmetro seja passado para a função. Desta maneira para utilização de métodos de tipos específicos de dados é necessária uma declaração de função. Exemplo:

const vetor = ["a","b","c"];

const toUpper = (str) => str.toUpperCase();

const maiusculas = vetor.map(toUpper);

No código acima tínhamos um vetor de caracteres que para passar para maiúsculo definimos a função *toUpper*, que recebe uma string e aplica o método *toUpperCase*.

Filter

Função de alta ordem semelhante ao MAP, sua diferença é que ele é utilizado para filtrar o vetor. Portanto a função do filter é receber um elemento e retornar um booleano. Por exemplo, há um vetor numérico e queremos dele somente os números pares.

const vetor = [1,2,3,4,5,6,7,8,9,10];

const pares = vetor.filter(x => x % 2 === 0);

Essa função retornará um vetor com apenas os números pares.

Reduce

Utilizado para reduzir o vetor a um valor ou objeto, como por exemplo somar os valores do vetor e reduzir a um único valor. Pode se colocar o valor inicial da variável que recebe a soma ao final da função, mas não é uma regra.

const vetor = [1,2,3,4,5,6];

const soma = vetor.reduce((estado, item) => estado + item, 0);

É necessário passar uma variável que irá armazenar a evolução do estado conforme o código é executado, é uma espécie de acumulador que vai guardando a soma parcial até o elemento presente, e outra variável que irá receber o próximo elemento a ser somado.

É possível utilizar a function map junto com o reduce. Por exemplo um carrinho de compras possuí além do id do produto, seu preço e sua quantidade. Para se ter um total necessitamos multiplicar o preço do item pela sua quantidade, é ai que entra a function map, onde já entregamos esse valor atualizado para um novo vetor que irá colocar o preço total de cada item a depender de sua quantidade.

**const** carrinho = [

    {id: 1, preco: 2, qtd: 2},

    {id: 2, preco: 3, qtd: 1}

]

**const** subtotal = item **=>** item.preco \* item.qtd;

**const** total = carrinho

                .map(subtotal)

                .reduce((soma, subtotal) **=>** subtotal + soma, 0)

console.log(total);

**Bibliografia**

<https://app.becas-santander.com/pt/program/bolsas-santander-tecnologia-santander-coders-web-full-stack-2021>

<https://natahouse.com/pt/hoisting>

<https://dev.to/joaoava/tipagem-fraca-forte-dinamica-e-estatica-g8k>

<https://robsoncastilho.com.br/2019/11/16/linguagens-estaticamente-ou-dinamicamente-tipadas/>

<https://youtu.be/Mbwg0YIZwYo?list=PLnNURxKyyLIIyJ_XKZHzU8SQyGu7yuqhn>

<https://www.youtube.com/watch?v=q5T_VMpvQyk>

<https://www.youtube.com/watch?v=9cNRNlhb-EA>

<https://www.youtube.com/watch?v=y8otiWoVfRI>

<https://www.youtube.com/watch?v=HFG_p4K2MAc>

<https://www.youtube.com/watch?v=8SJwvcHD1ns>

<https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array>

<https://www.youtube.com/watch?v=S5Mn0qQzJ-0>

<https://www.youtube.com/watch?v=PKnpSaAQdd4>