JavaScript Completo ES6

**Hoisting:** Ao criar variáveis e funções, sua declaração sempre é movida para o começo do código.

**Tipos de dados:**

var nome = 'André'; // String

var idade = 28; // Number

var possuiFaculdade = true; // Boolean

var time; // Undefined

var comida = null; // Null

var simbolo = Symbol() // Symbol

var novoObjeto = {} // Object

**Para ver o tipo de dados, pode utilizar typeof:**

var nome = 'Reydson';

console.log(typeof nome); //retorna “string”

Devido a um bug, o typeof de null é object.

**Template string:**

`O nome é ${nome}`

**Podemos utilizar o "e" para definir a quantidade de zeros ou de casas decimais:**

var exp = 2e10; //20000000000

exp = 2e-2; //0.02

**Para checar se um valor é Not a Number, existe a função isNaN()**

**Existem operadores unários que convertem strings em números:**

console.log(typeof '10'); //string

console.log(typeof +'10'); //number

console.log(typeof -'10'); //number

**Valores considerados como falso:**

if(false)

if(0)

if(NaN)

if(null)

if(undefined)

if('')

**Qualquer coisa diferente dos anteriores será considerado verdadeiro, exemplos:**

if(true)

if(1)

if(' ')

if('andre')

if(-5)

if({})

**O operador && retorna o primeiro valor falso que encontrar ou o último verdadeiro:**

true && true; // true

true && false; // false

false && true; // false

'Gato' && 'Cão'; // 'Cão'

(5 - 5) && (5 + 5); // 0

'Gato' && false; // false

(5 >= 5) && (3 < 6); // true

**Já o operador || retorna o primeiro valor verdadeiro ou o último valor falso:**

true || true; // true

true || false; // true

false || true; // true

'Gato' || 'Cão'; // 'Gato'

(5 - 5) || (5 + 5); // 10

'Gato' || false; // Gato

(5 >= 5) || (3 < 6); // true

**Podemos passar funções como argumentos para funções.**

**Dentro de objetos, as funções podem ser declaradas de duas formas:**

var quadrado = {

qtdLados: 4,

perimetro: function(lado) {

return lado \* this.qtdLados;

}

}

var quadrado2 = {

qtdLados: 4,

perimetro(lado) {

return lado \* this.qtdLados;

}

}

**Algumas funções presentes em strings e números:**

var nome = 'Reydson';

console.log(nome.length); //7

console.log(nome.toLocaleLowerCase()); //reydson

console.log(nome.charAt(2)); //y

console.log(nome.replace('ydson', 'inaldo')); //Reinaldo

var altura = 1.75;

console.log(altura.toString()); //'1.75'

console.log(altura.toFixed()); //2 (string)

console.log(altura.toFixed(4)); //1.7500 (string)

**Alguns métodos de Array:**

array.pop() Retorna o último elemento do array e o remove dele.

array.push(item) Adiciona o item no final do array e retorna a quantidade de itens.

**Var vs const e let:**

O var vaza o escopo de bloco:

if(true) {

var mes = 'Janeiro';

console.log(mes); //Janeiro

}

console.log(mes); //Janeiro

**Métodos para selecionar objetos:**

getElementById

getElementsByClassName

getElementsByTagName

querySelector

querySelectorAll

**Transformar elemento array-like em Array:**

Const array = Array.from(arrayLike);

**Lidando com as classes de um elemento do DOM:**

const menu = document.querySelector('.menu');

menu.classList; // lista de classes

menu.classList.add('ativo');

menu.classList.add('ativo', 'mobile'); // duas classes

menu.classList.remove('ativo');

menu.classList.toggle('ativo'); // adiciona/remove a classe

menu.classList.contains('ativo'); // true ou false

menu.classList.replace('ativo', 'inativo');

Forma menos usual:

menu.className; // string, retorna todas as classes separadas por espaço

menu.className += ' teste'; //também funciona

**Lidando com atributos de um elemento do DOM:**

const img = document.querySelector('img');

console.log(img.getAttribute('src'));

img.setAttribute('src', 'img/imagem2.jpg');

img.hasAttribute('id'); // true / false

img.removeAttribute('alt'); // remove o alt

Forma menos usual:

const animais = document.querySelector('.animais');

console.log(animais.attributes[0]); //mostra o primeiro atributo

console.log(animais.attributes.class.value); //mostra o valor do atributo class

console.log(animais.attributes['class'].value); //mostra o valor do atributo class

console.log(animais.attributes.id.value); //mostra o valor do atributo id

console.log(animais.attributes['id'].value); //mostra o valor do atributo id

**Pegando algumas medidas:**

const listaAnimais = document.querySelector('.animais-lista');

console.log(listaAnimais.clientHeight); //altura visível

console.log(listaAnimais.scrollHeight); //altura incluindo o que está oculto pela rolagem

console.log(listaAnimais.clientWidth); //largura

console.log(listaAnimais.offsetTop); //distância até o topo da página

console.log(listaAnimais.offsetLeft); //distância até o lado esquerdo

**Outra forma:**

const rect = listaAnimais.getBoundingClientRect(); //retorna um objeto com várias medidas

console.log(rect);

console.log(window.innerWidth); //largura da página exibida

console.log(window.outerWidth); //largura da janela do navegador inteira

console.log(window.innerHeight); //altura da página exibida

console.log(window.outerHeight); //altura da janela do navegador inteira

console.log(window.scrollY); //o quanto já foi rolado verticalmente

console.log(window.scrollX); //o quanto já foi rolado horizontalmente

**Utilizando media queries:**

const small = window.matchMedia('(max-width: 600px)').matches;

if(small) {

console.log('A tela é menor que 600px');

} else {

console.log('A tela é maior que 600px');

}

**Alguns atributos do evento:**

function executarCallback(event) {

const currentTarget = event.currentTarget; // elemento onde o evento foi adicionado

const target = event.target; // elemento que o clique ocorreu

const type = event.type; // tipo de evento, retorna uma string

const path = event.path; // elementos indo no elemento clicado até o objeto window

console.log(currentTarget, target, type, path);

}

**Para anular a ação que disparou o evento:**

event.preventDefault();

**Dentro da função de call-back (criada com function) do evento, o this corresponde ao event.currentTarget, ou seja, o elemento em que o evento foi adicionado.**

**Exemplo com eventos de teclas:**

function handleKeyboard(event) {

if(event.key === 'a') {

console.log('Apertou a tecla aaaaaa');

} else {

console.log(`Apertou a tecla ${event.key}`);

}

}

window.addEventListener('keydown', handleKeyboard);

**Transversing (navegação pelo DOM):**

const lista = document.querySelector('.animais-lista');

lista.parentElement; // pai

lista.parentElement.parentElement; // pai do pai

lista.previousElementSibling; // elemento acima

lista.nextElementSibling; // elemento abaixo

lista.children; // HTMLCollection com os filhos

lista.children[0]; // primeiro filho

lista.children[--lista.children.length]; // último filho

lista.querySelectorAll('li'); // todas as LI's

lista.querySelector('li:last-child'); // último filho

**Movendo elementos do DOM:**

contato.appendChild(lista); // move lista para o final de contato

contato.insertBefore(lista, titulo); // insere a lista antes de titulo

contato.removeChild(titulo); // remove título de contato

contato.replaceChild(lista, titulo); // substitui título por lista, sendo o título descartado

**Criando novos elementos:**

const animais = document.querySelector('.animais');

const novoH1 = document.createElement('h1');

novoH1.innerText = 'Novo Título';

novoH1.classList.add('titulo');

animais.appendChild(novoH1);

**Clonar elementos:**

const titulo = document.querySelector('h1');

const titulo2 = document.querySelector('h1');

const novoTitulo = titulo;

// titulo, titulo2 e novoTitulo são iguais

const cloneTitulo = titulo.cloneNode(true); //O true indica que tudo que estiver dentro do h1 também será clonado

const contato = document.querySelector('.contato');

contato.appendChild(cloneTitulo);

**Temos 3 formas de criar um método dentro de um objeto:**

const objeto = {

funcao1: function() {

console.log('Função 1');

},

funcao2: () => {

console.log('Função 2');

},

funcao3() {

console.log('Função 3');

}

}

**Funções construtoras:**

function Carro(marca, preco) {

const taxa = 1.2;

this.marca = marca;

this.preco = preco \* taxa;

this.andar = function() {

console.log('Andou');

}

}

const fiat = new Carro('Fiat', 1000);

**Posso utilizar diretamente o valor passado por parâmetro:**

function Dom(seletor) {

const element = document.querySelector(seletor);

this.ativo = function(classe) {

element.classList.add(classe);

};

}

**Podemos acessar uma função do prototype (pouco utilizado):**

const lista = document.querySelectorAll('li');

// Transforma em uma array

const listaArray = Array.prototype.slice.call(lista);

**Native:** Objetos que existem em qualquer implementação do JavaScript, tem no Navegador e no Node, por exemplo.

**Host:** Objetos que existem apenas naquela implementação, QuerySelectorAll, por exemplo.

**User:** Objetos criados pelo programador.

**Checar se um objeto existe:**

const existe = typeof teste !== 'undefined';

**Métodos de String:**

const nome = 'Reydson Barros';

nome.length //14

nome[1] //e

nome.charAt(1) //e

nome.concat(' abc', ' abc') // Reydson Barros abc abc

nome.includes('ey')) // true (pode receber um segundo parâmetro que determina a partir de qual posição irá procurar)

nome.startsWith('Rey') //true

nome.endsWith('son') //true

nome.slice(1, 5) //eyds (posição do primeiro e do último caracter sem incluir ele)

nome.substring(1, 5) //eyds (mesma coisa do anterior)

nome.indexOf('o') //5 (primeira ocorrência)

nome.lastIndexOf('o') //12 (última ocorrência)

const preco = 'R$ 99,00';

console.log(preco.padStart(12, '.')) //....R$ 99,00 (adiciona até completar a qtd de caracteres)

console.log(preco.padEnd(12, '.')) //R$ 99,00.... (adiciona até completar a qtd de caracteres)

const palavra = 'Ta';

console.log(palavra.repeat(5)); //TaTaTaTaTa

let listaItens = 'Camisas Bonés Calças Bermudas Vestidos Saias';

listaItens = listaItens.replace(/[ ]+/g, ', '); //Camisas, Bonés, Calças, Bermudas, Vestidos, Saias

let listaItens2 = 'Camisas Bonés Calças Bermudas Vestidos Saias';

listaItens2 = listaItens2.replace(' ', ', '); //Camisas, Bonés Calças Bermudas Vestidos Saias (só substitui o primeiro)

let preco2 = 'R$ 1200,43';

preco2 = preco2.replace(',', '.'); // 'R$ 1200.43'

const listaNova = 'Camisas Bonés Calças Bermudas Vestidos Saias';

const listaArray = listaNova.split(' ');

console.log(listaArray); //(6) ['Camisas', 'Bonés', 'Calças', 'Bermudas', 'Vestidos', 'Saias']

**Trocando uma tag:**

const htmlText = '<div>O melhor item</div><div>A melhor lista</div>';

const htmlArray = htmlText.split('div');

const htmlNovo = htmlArray.join('section');

console.log(htmlNovo); //<section>O melhor item</section><section>A melhor lista</section>

console.log('Teste'.toLowerCase()); //teste

console.log('Teste'.toUpperCase()); //TESTE

console.log(' TESTE '.trim()); //"TESTE"

console.log(' TESTE '.trimStart()); //"TESTE "

console.log(' TESTE '.trimEnd()); //" TESTE"

**Métodos de Number:**

Number.isNaN(NaN); //true

Number.isNaN(10); //false

Number.isInteger(10); //true

Number.isInteger(10.5); //false

console.log(Number.parseInt(10.5)); //10

console.log(Number.parseInt(10.9)); //10

console.log(Number.parseInt("10.5")); //10

console.log(Number.parseFloat("10.9")); //10.9

console.log(Number.parseFloat("10.11abcd")); //10.11

console.log(Number.parseFloat("$10.11")); //NaN

console.log(Number.parseFloat(" 10.11")); //10.11

**Permite definir a quantidade de casas decimais, arredonda para cima a partir de 0.5 (retorna uma string):**

const preco = 2.99;

console.log(preco.toFixed()); //3

const carro = 1000.456;

console.log(carro.toFixed(2)); // 1000.46

const preco2 = 1499.49;

console.log(preco2.toFixed()); // 1499

const preco = 59.49;

preco.toLocaleString('en-US', {style: 'currency', currency: 'USD'}); // $59.49

preco.toLocaleString('pt-BR', {style: 'currency', currency: 'BRL'}); // R$ 59,49

**Objeto Math:**

Math.PI // 3.14159

Math.E // 2.718

Math.LN10 // 2.303

abs() retorna o valor absoluto, ou seja, transforma negativo em positivo. ceil() arredonda para cima, retornando sempre uma integral e floor para baixo. round() arredonda para o número integral mais próximo.

Math.abs(-5.5); // 5.5

Math.ceil(4.8334); // 5

Math.ceil(4.3); // 5

Math.floor(4.8334); // 4

Math.floor(4.3); // 4

Math.round(4.8334); // 5

Math.round(4.3); // 4

max() retorna o maior número de uma lista de argumentos, min() o menor número e random() um número aleatório entre 0 e 1.

Math.max(5,3,10,42,2); // 42

Math.min(5,3,10,42,2); // 2

Math.random(); // 0.XXX

Math.round(Math.random() \* 100); // entre 0 e 100

Math.round(Math.random() \* 500); // entre 0 e 500

// Número random entre 72 e 32

Math.floor(Math.random() \* (72 - 32 + 1)) + 32;

Math.floor(Math.random() \* (max - min + 1)) + min;

**Arrays:**

Array.isArray(''); //false

Array.isArray([]); //true

**Algumas formas de criar um array:**

Array.of(10); // [10]

Array.of(1,2,3,4); // [1,2,3,4]

new Array(5); // [,,,,]

Array(5); // [,,,,]

Array(1,2,3,4); // [1,2,3,4]

**O método sort() põe o array em ordem alfabética (modifica ele):**

const instrumentos = ['Guitarra', 'Baixo', 'Violão'];

instrumentos.sort();

instrumentos; // ['Baixo', 'Guitarra', Violão]

const idades = [32,21,33,43,1,12,8];

idades.sort();

idades; // [1, 12, 21, 32, 33, 43, 8] **ORDEM ERRADA!**

**[].unshift() adiciona elementos ao início da array e retorna o length da mesma. [].push() adiciona elementos ao final da array e retorna o length da mesma.**

const carros = ['Ford', 'Fiat', 'VW'];

carros.unshift('Honda', 'Kia'); // 5

carros; // ['Honda', 'Kia', 'Ford', 'Fiat', 'VW'];

carros.push('Ferrari'); // 6

carros; // ['Honda', 'Kia', 'Ford', 'Fiat', 'VW', 'Ferrari'];

**[].shift() remove o primeiro elemento da array e retorna o mesmo. [].pop() remove o último elemento da array e retorna o mesmo.**

const carros = ['Ford', 'Fiat', 'VW', 'Honda'];

const primeiroCarro = carros.shift(); // 'Ford'

carros; // ['Fiat', 'VW', 'Honda'];

const ultimoCarro = carros.pop(); // 'Honda'

carros; // ['Fiat', 'VW'];

**[].reverse() inverte os itens da array e retorna a nova array (modifica o array).**

const carros = ['Ford', 'Fiat', 'VW', 'Honda'];

carros.reverse(); // ['Honda', 'VW', 'Fiat', 'Ford'];

**[].splice(index, remover, item1, item2, ...) adiciona valores na array a partir do index. Remove a quantidade de itens que for passada no segundo parâmetro (retorna esses itens).**

const carros = ['Ford', 'Fiat', 'VW', 'Honda'];

carros.splice(1, 0, 'Kia', 'Mustang'); // []

carros; // ['Ford', 'Kia', 'Mustang', 'Fiat', 'VW', 'Honda']

carros.splice(3, 2, 'Ferrari'); // ['Fiat', 'VW']

carros; // ['Ford', 'Kia', 'Mustang', 'Ferrari', 'Honda']

**[].fill(valor, inicio, final) preenche a array com o valor, do início até o fim.**

['Item1', 'Item2', 'Item3', 'Item4'].fill('Banana');

// ['Banana', 'Banana', 'Banana', 'Banana']

['Item1', 'Item2', 'Item3', 'Item4'].fill('Banana', 2); // ['Item1', 'Item2', 'Banana', 'Banana'] (a partir do index 2)

['Item1', 'Item2', 'Item3', 'Item4'].fill('Banana', 1, 3); // ['Item1', 'Banana', 'Banana', 'Item4'] (do index 1 até antes do index 3)

**[].concat() irá concatenar a array com o valor passado, os arrays não são modificados:**

const transporte1 = ['Barco', 'Aviao'];

const transporte2 = ['Carro', 'Moto'];

const transportes = transporte1.concat(transporte2);

// ['Barco', 'Aviao', 'Carro', 'Moto'];

const maisTransportes = [].concat(transporte1, transporte2, 'Van');

// ['Barco', 'Aviao', 'Carro', 'Moto', 'Van'];

**[].includes(valor) verifica se a array possui o valor e retorna true ou false. [].indexOf(valor) verifica se a array possui o valor e retorna o index do primeiro valor na array. Já o [].lastIndexOf(valor) retorna o index do último. (Se não tiver no array, retorna -1)**

const linguagens = ['html', 'css', 'js', 'php', 'python', 'js'];

linguagens.includes('css'); // true

linguagens.includes('ruby'); // false

linguagens.indexOf('python'); // 4

linguagens.indexOf('js'); // 2

linguagens.lastIndexOf('js'); // 5

**[].join(separador) junta todos os valores da array e retorna uma string com eles. Se você passar um valor como parâmetro, este será utilizado durante a junção de cada item da array.**

const linguagens = ['html', 'css', 'js', 'php', 'python'];

linguagens.join(); // 'html,css,js,php,python'

linguagens.join(' '); // 'html css js php python'

linguagens.join('-\_-'); // 'html-\_-css-\_-js-\_-php-\_-python'

**[].slice(inicio, final) retorna os itens começando pelo início e indo até o valor de final.**

const linguagens = ['html', 'css', 'js', 'php', 'python'];

linguagens.slice(3); // ['php', 'python']

linguagens.slice(1, 4); // ['css', 'js', 'php']

const cloneLinguagens = linguagens.slice(); **//ele também pode ser utilizado para clonar um array**

**Array e iteração:**

**[].forEach(callback(itemAtual, index, array)) a função de callback é executada para cada item da array. Ela possui três argumentos, itemAtual (valor do item da array), index (index do valor na array) e array (array original).**

const carros = ['Ford', 'Fiat', 'Honda'];

carros.forEach(function(item, index, array) {

console.log(item.toUpperCase());

});

**[].map(callback(itemAtual, index, array)) retorna uma nova array com valores atualizados de acordo com o return de cada iteração.**

const carros = ['Ford', 'Fiat', 'Honda'];

const newCarros = carros.map((item) => {

return 'Carro ' + item;

});

carros; // ['Ford', 'Fiat', 'Honda']

newCarros; // ['Carro Ford', 'Carro Fiat', 'Carro Honda'];

**[].reduce(callback(acumulador, valorAtual, index, array), valorInicial) executa a função de callback para cada item da Array. Um valor especial existe nessa função de callback, ele é chamado de acumulador, mas é na verdade apenas o retorno da iteração anterior.**

const aulas = [10, 25, 30];

const total1 = aulas.reduce((acumulador, atual) => {

return acumulador + atual;

});

total1; // 65

const total2 = aulas.reduce((acc, cur) => acc + cur, **100**);

total2; // 165

**Se não passar esse valor inicial, mesmo que seja zero, ele pula a primeira iteração, nesse caso, o acumulador já começa como sendo o primeiro elemento do array.**

**Também pode utilizar reduce para objetos:**

const resultado = aulas2.reduce((acumulador, item, index) => {

acumulador[index] = item.nome;

return acumulador;

}, {});

console.log(resultado); //{0: 'HTML 1', 1: 'HTML 2', 2: 'CSS 1', 3: 'JS 1'}

**[].REDUCERIGHT()**

**Existe também o método [].reduceRight(), a diferença é que este começa a iterar da direita para a esquerda, enquanto o reduce itera da esquerda para a direita.**

const frutas = ['Banana', 'Pêra', 'Uva'];

const frutasRight = frutas.reduceRight((acc, fruta) => acc + ' ' + fruta);

const frutasLeft = frutas.reduce((acc, fruta) => acc + ' ' + fruta);

frutasRight; // Uva Pêra Banana

frutasLeft; // Banana Pêra Uva

**[].some(), se pelo menos um return da iteração for truthy, ele retorna true.**

const frutas = ['Banana', 'Pêra', 'Uva'];

const temUva = frutas.some((fruta) => {

return fruta === 'Uva';

}); // true

function maiorQue100(numero) {

return numero > 100;

}

const numeros = [0, 43, 22, 88, 101, 2];

const temMaior = numeros.some(maiorQue100); // true

**[].every(), se todos os returns das iterações forem truthy, o método irá retornar true. Se pelo menos um for falsy, ele irá retornar false.**

const frutas = ['Banana', 'Pêra', 'Uva', ''];

// False pois pelo menos uma fruta

// está vazia '', o que é um valor falsy

const arraysCheias = frutas.every((fruta) => {

return fruta; // false

});

const numeros = [6, 43, 22, 88, 101, 29];

const maiorQue3 = numeros.every(x => x > 3); // true

**[].find(), retorna o valor atual da primeira iteração que retornar um valor truthy. Já o [].findIndex(), ao invés de retornar o valor, retorna o index deste valor na array.**

const frutas = ['Banana', 'Pêra', 'Uva', 'Maçã'];

const buscaUva = frutas.findIndex((fruta) => {

return fruta === 'Uva';

}); // 2

const numeros = [6, 43, 22, 88, 101, 29];

const buscaMaior45 = numeros.find(x => x > 45); // 88

**[].filter(), retorna uma array com a lista de valores que durante a sua iteração retornaram um valor truthy.**

const frutas = ['Banana', undefined, null, '', 'Uva', 0, 'Maçã'];

const arrayLimpa = frutas.filter((fruta) => {

return fruta;

}); // ['Banana', 'Uva', 'Maçã']

const numeros = [6, 43, 22, 88, 101, 29];

const buscaMaior45 = numeros.filter(x => x > 45); // [88, 101]

**O filter também pode ser utilizado em objetos:**

const aulas = [

{

nome: 'HTML 1',

min: 15

},

{

nome: 'HTML 2',

min: 10

},

{

nome: 'CSS 1',

min: 20

},

{

nome: 'JS 1',

min: 25

},

]

const aulasMaiores = aulas.filter((aula) => {

return aula.min > 15;

});

// [{nome: 'CSS 1', min: 20}, {nome: 'JS 1', min: 25}]

**Podemos criar uma função utilizando seu construtor, onde o primeiro parâmetro consiste nos argumentos da função e o segundo consiste em seu código:**

const perimetroQuadrado = new Function('lado', 'return lado \* 4');

**A toda função possui um método call que pode ser utilizado para executá-la, isso é bem útil para objetos array-like:**

const lis = document.querySelectorAll('li');

const lis2 = Array.prototype.filter.call(lis, (element) => {

return +element.innerText % 2 == 0;

}); **//O primeiro parâmetro é o this da função (objeto que ela operará)**

**Existe também o apply, que ao invés de receber os argumentos um a um, recebe um array de argumentos, o que pode ser útil:**

const numeros = [3,4,6,1,34,44,32];

Math.max.apply(null, numeros);

**Existe ainda o bind, que retorna a função no contexto passado, mas não a ativa:**

const li = document.querySelectorAll('li');

const filtrarLi = Array.prototype.filter.bind(li, function(item) {

return item.classList.contains('ativo');

});

filtrarLi();**//Pode passar argumentos tanto aqui, quanto na hora de aplicar o bind, podendo ser 2 em cada, por exemplo**

**Object.create(obj, defineProperties):** retorna um novo objeto que terá como protótipo o objeto do primeiro argumento. Exemplo:

const carro = {

rodas: 4,

marca: "Marca",

acelerar() {

return `${this.marca} acelerou as ${this.rodas} rodas`;

},

};

const honda = Object.create(carro);

honda.marca = "Honda";

console.log(honda.acelerar());

**Object.assign(alvo, obj1, obj2)** adiciona ao alvo as propriedades e métodos enumeráveis dos demais objetos passados. O assign irá modificar o objeto alvo.

**Object.defineProperties(alvo, propriedades)** adiciona ao alvo novas propriedades. A diferença aqui é a possibilidade de serem definidas as características dessas propriedades, definindo-a como uma “constante”, por exemplo. Exemplo:

const moto = {}

Object.defineProperties(moto, {

rodas: {

value: 2,

configurable: false, // impede deletar e mudança de valor

enumarable: true, // torna enumerável

},

capacete: {

value: true,

configurable: true,

writable: false, // impede mudança de valor

},

})

moto.rodas = 4;

delete moto.capacete;

moto;

// {rodas: 2}

**GET E SET**

É possível definirmos diferentes comportamentos para get e set de uma propriedade. Lembrando que ao acionarmos uma propriedade obj.propriedade, a função get é ativada e ao setarmos obj.propriedade = 'Valor' a função de set é ativada.

const moto = {}

Object.defineProperties(moto, {

velocidade: {

get() {

return this.\_velocidade;

},

set(valor) {

this.\_velocidade = 'Velocidade ' + valor;

}

},

})

moto.velocidade = 200;

moto.velocidade;

// Velocidade 200

**OBJECT.FREEZE(), OBJECT.SEAL(), OBJECT.PREVENTEXTENSIONS()**

**Object.freeze()** impede qualquer mudança nas propriedades. **Object.seal()** previne a adição de novas propriedades e impede que as atuais sejam deletadas. **Object.preventExtensions()** previne a adição de novas propriedades.

const carro = {

marca: 'Ford',

ano: 2018,

}

Object.freeze(carro);

Object.seal(carro);

Object.preventExtensions(carro);

Object.isFrozen(carro); // true

Object.isSealed(carro); // true

Object.isExtensible(carro); // true

**A função toString() pode ser utilizada para descobrir o tipo de um objeto da seguinte forma:**

**Object.prototype.toString.call(valor)**

const frutas = ['Banana', 'Uva'];

frutas.toString(); // 'Banana,Uva'

typeof frutas; // object

Object.prototype.toString.call(frutas); // [object Array]

const frase = 'Uma String';

frase.toString(); // 'Uma String'

typeof frase; // string

Object.prototype.toString.call(frase); // [object String]

const carro = {marca: 'Ford'};

carro.toString(); // [object Object]

typeof carro; // object

Object.prototype.toString.call(carro); // [object Object]

const li = document.querySelectorAll('li');

typeof li; // object

Object.prototype.toString.call(li); // [object NodeList]

**Dataset:**

<div data-cor="azul" data-width="500">Uma Div</div>

Para selecionar utilizando o atributo, pode-se fazer o seguinte:

let div = document.querySelector('div');

div = document.querySelector('[data-cor]');

div = document.querySelector('[data-cor="azul"]');

div.dataset; // DOMStringMap {cor: "azul", width: "500"}

div.dataset.cor; // 'azul'

div.dataset.width; // '500'

Para deletar:

delete div.dataset.cor;

Exemplo de uso mais prático:

<div data-anima="left" data-tempo="1000">Div 1</div>

<div data-anima="right" data-tempo="2000">Div 2</div>

const divs = document.querySelectorAll('[data-anima]');

divs.forEach((div) => {

div.classList.add(div.dataset.anima);

});

Obs.: data-anima-scroll vira elemento.dataset.animaScroll (Camel case)

**Módulos:**

Vantagens:

* Manutenção: Dividir o código em diferentes arquivos com funções específicas (módulos) facilita a manutenção.
* Compartilhamento: O compartilhamento de código com outros projetos é facilitado, pois basta você importar um módulo específico.
* Nativo no ES6+: Ferramentas que permitem dividirmos o código em módulos já existem a bastante tempo. Grunt, Gulp, Webpack, Browserify, Parcel e outras. Mas agora os módulos são nativos.

Para utilizar, basta adicionar type="module" na tag script do HTML e utilizar a a palavra chave export na frente do valor que deseja exportar (usando default se for único). E import nome from arquivo.js para importar.

<script type="module" src="js/script.js"></script>

// arquivo scroll-suave.js

export default function scrollSuave() {

...

};

// arquivo script.js

import scrollSuave from './scroll-suave.js';

scrollSuave();

**Named exports:** Além do default, pode também exportar com o nome, mas na hora de importar, tem que especificar esse nome.

// arquivo scroll.js

export function scrollSuave() {

...

};

export function scrollAnimacao() {

...

};

// arquivo script.js

import { scrollSuave, scrollAnimacao } from './scroll.js';

scrollSuave();

scrollAnimacao();

Pode também importar todos ao mesmo tempo e ainda definindo um apelido:

import \* as scroll from './scroll.js';

scroll.scrollSuave();

scroll.scrollAnimacao();

Além de funções, pode exportar também constantes, objetos, strings, classes, etc.:

// arquivo configuracao.js

export function scrollSuave() {};

export const ano = 2000;

export const obj = {nome: 'Ford'};

export const str = 'Frase';

export class Carro {};

// arquivo script.js

import \* as conf from './configuracao.js';

conf.str;

conf.obj;

conf.ano;

**Use strict:**

O modo estrito previne algumas ações consideradas como erro. Basta adicionarmos 'use strict' no topo de um arquivo, que ele entrará neste modo:

'use strict';

Alguns exemplos de restrições:

nome = 'Ford'; // erro, variável global (sem var, let ou const)

delete Array.prototype; // erro, não deletável

window.top = 200; // erro, não pode mudar

const arguments = 3.14; // escrever em palavra reservada

**Além de uma função, podemos passar também um objeto para um evento:**

const onMouseLeave = {

tooltipBox: '',

element: '',

handleEvent() {

this.tooltipBox.remove();

this.element.removeEventListener('mouseleave', this);

}

}

element.addEventListener('mouseleave', onMouseLeave);

**Podemos também remover um evento:**

element.removeEventListener('click', funcaoDeCallback); //precisa passar os dois para que encontre o evento correto

**Event bubble:** Após executar um evento em um elemento, ele executa esse evento nos elementos pais também.

**SETTIMEOUT()**

setTimeout(callback, tempo, arg1, arg2, ...) método assíncrono que ativa o callback após o tempo. Não existe garantia de que o código será executado exatamente após o tempo, pois o callback entra na fila e espera pela Call Stack estar vazia. Exemplo:

function espera(texto) {

console.log(texto);

}

setTimeout(espera, 1000, 'Depois de 1s'); (O 'Depois de 1s' será passado como argumento da função de callback)

Se não passarmos o argumento de tempo ele irá assumir o valor 0 e entrará na fila imediatamente para ser executado. Podemos passar uma função anônima diretamente com argumento. Examplo:

setTimeout(() => {

console.log('Após 0s?');

});

**THIS E WINDOW**

setTimeout é um método do objeto Window. O valor de this dentro do mesmo callback é uma referência ao seu objeto no caso Window.

const btn = document.querySelector('button');

btn.addEventListener('click', handleClick);

function handleClick(event) {

setTimeout(function() {

this.classList.add('active');

}, 1000)

}

// Erro pois window.classList não existe

**ARROW FUNCTION**

Quando utilizamos uma Arrow Function como callback, o contexto de this passa a ser do local onde o setTimeout foi iniciado.

const btn = document.querySelector('button');

btn.addEventListener('click', handleClick);

// this agora é btn.

function handleClick(event) {

setTimeout(() => {

this.classList.add('active');

}, 1000)

}

**SETINTERVAL**

setInterval(callback, tempo, arg1, arg2, ...), irá ativar o callback toda vez que a quantidade de tempo passar.

function loop(texto) {

console.log(texto);

}

setInterval(loop, 1000, 'Passou 1s');

// loop a cada segundo

let i = 0;

setInterval(() => {

console.log(i++);

}, 1000);

**CLEARINTERVAL**

clearInterval(var), podemos parar um intervalo com o clearInterval. Para isso precisamos atribuir o setInterval a uma variável.

const contarAte10 = setInterval(callback, 1000);

let i = 0;

function callback() {

console.log(i++);

if (i > 10) {

clearInterval(contarAte10);

}

}

**MutationObserver:** Serve para monitorar as modificações em um objeto, como a mudança em atributos, por exemplo.

Exemplo:

function handleMutation(mutation) {

if(mutation[0].target.classList.contains('ativo')) {

observer.disconnect()

animaNumeros();

}

}

const observerTarget = document.querySelector('.numeros');

const observer = new MutationObserver(handleMutation);

observer.observe(observerTarget, {attributes: true});

**Date:** Objeto para a manipulação de datas.

const agora = new Date();

agora;

// Semana Mês Dia Ano HH:MM:SS GMT

agora.getDate() // Dia

agora.getDay() // Dia da Semana ex: 5 = Fri (começa em 0 = domingo)

agora.getMonth() // Número dia mês

agora.getFullYear() // Ano

agora.getHours() // Hora

agora.getMinutes() // Minutos

agora.getTime() // ms desde 1970

agora.getUTCHours() - 3 // Brasília

Também é possível passar uma data no construtor:

const futuro = new Date('Dec 24 2026');

Com o timestamp dá para calcular facilmente a **quantidade de dias entre datas**:

const agora = new Date();

const promocao = new Date('December 24 2018 23:59');

function converterEmDias(time) {

return time / (24 \* 60 \* 60 \* 1000);

}

const diasAgora = converterEmDias(agora);

const diasPromocao = converterEmDias(promocao);

const faltam = diasPromocao - diasAgora;

**Formulários:** é possível pegar os formulários da página utilizando o atributo forms do document:

document.forms; // lista com os formulários

document.forms.contato; // form com nome contato

document.forms.contato.elements; // elementos

document.forms[0].elements[0].value; // valor do primeiro campo do primeiro formulário

**Pode buscar também pelo nome ou id do input:**

const nome = document.forms[0].elements.nome.value;

A propriedade value retorna o valor do elemento no formulário. Se adicionarmos um callback ao keyup (tecla levantar), podemos ficar de olho no evento e puxar o valor sempre que ele mudar. change dispara quando houver mudanças.

const form = document.getElementById('contato');

function handleKeyUp(event) {

console.log(event.target.value); //pega o valor do campo que está sendo editado

}

form.addEventListener('keyup', handleKeyUp)

**VALIDAÇÃO**

O método **checkValidity** verifica se um input com o atributo required, é válido ou não. A propriedade **validationMessage** possui a mensagem padrão de erro do browser. É possível modicar com **setCustomValidity('')**

<input type="email" name="email" id="email" required>

<span class="erro"></span>

const form = document.getElementById('contato');

function handleChange(event) {

const target = event.target;

if(!target.checkValidity()) {

target.classList.add('invalido');

target.nextElementSibling.innerText = target.validationMessage;

} else {

target.classList.remove('invalido');

}

}

form.addEventListener('change', handleChange);

**PEGANDO TODOS OS VALORES**

Ao invés de selecionarmos elemento por elemento, podemos utilizar um objeto para colocarmos todos os dados que o usuário colocar no formulário.

<form name="contato" id="contato">

<label for="nome">Nome</label>

<input type="text" name="nome" id="nome">

<label for="email">Email</label>

<input type="email" name="email" id="email">

<label for="mensagem">Mensagem</label>

<textarea name="mensagem" id="mensagem"></textarea>

</form>

const form = document.getElementById('contato');

const dados = {};

function handleChange(event) {

dados[event.target.name] = event.target.value;

}

form.addEventListener('change', handleChange);

**Acessando atributos e métodos através do uso de colchetes:**

botao.name = "abc";

botao.[ "name "] = "abc";

botao.remove()

botao['remove']()

**Local storage e session storage:**

O session storage armazena apenas naquela sessão do browser e o local storage continua armazenando mesmo que feche o browser e reinicie o computador, por exemplo.

Para utilizar, basta armazenar ou utilizar os valores da seguinte forma:

localStorage.nome = 'Reydson';

sessionStorage.nome = 'Reydson';

**JavaScript Assíncrono:**

Exemplos: setTimeout, Ajax, Promises, Fetch, Async.

Os eventos assíncronos são executados após a finalização do código principal, isso ocorre para evitar bugs.

**Promises:**

Promise é uma função construtora de promessas. Existem dois resultados possíveis de uma promessa, ela pode ser resolvida, com a execução do primeiro argumento, ou rejeitada se o segundo argumento for ativado. Exemplo:

const promessa = new Promise(function(resolve, reject) {

let condicao = false;

if(condicao) {

resolve('Estou pronto!');

} else {

reject(Error('Um erro ocorreu.'));

}

});

console.log(promessa); // Promise {<rejected>: Error:...}

O poder das Promises está no método then() do seu protótipo. O Callback deste método só será ativado quando a promise for resolvida. O argumento do callback será o valor passado na função resolve.

promessa.then(function(resolucao) {

console.log(resolucao); // 'Estou pronto!'

});

**THEN().THEN()**

O método then() retorna outra Promise. Podemos colocar then() após then() e fazer um encadeamento de promessas. O valor do primeiro argumento de cada then, será o valor do retorno anterior.

**catch()**

O método catch(), do protótipo de Promises, adiciona um callback a promise que será ativado caso a mesma seja rejeitada.

const promessa = new Promise((resolve, reject) => {

let condicao = false;

if(condicao) {

resolve('Estou pronto!');

} else {

reject(Error('Um erro ocorreu.'));

}

});

promessa.then(resolucao => {

console.log(resolucao);

}).catch(reject => {

console.log(reject);

});

**THEN(RESOLVE, REJECT)**

Também pode passar um segundo callback no then para capturar a rejeição, isso cria o mesmo efeito do catch. Exemplo:

promessa.then(resolucao => {

console.log(resolucao);

}, reject => {

console.log(reject);

});

**FINALLY()**

finally() executará a função anônima assim que a promessa acabar. A diferença do finally é que ele será executado independente do resultado, se for resolvida ou rejeitada. Ele não recebe valor algum.

**PROMISE.ALL()**

Retornará uma nova promise assim que todas as promises dentro dela forem resolvidas ou pelo menos uma rejeitada. **A reposta é uma array com as respostas de cada promise.**

const login = new Promise(resolve => {

setTimeout(() => {

resolve('Login Efetuado');

}, 1000);

});

const dados = new Promise(resolve => {

setTimeout(() => {

resolve('Dados Carregados');

}, 1500);

});

const tudoCarregado = Promise.all([login, dados]);

tudoCarregado.then(respostas => {

console.log(respostas); // Array com ambas respostas

});

**PROMISE.RACE()**

Retornará uma nova promise assim que a primeira promise for resolvida ou rejeitada. Essa nova promise terá a resposta da primeira resolvida.

const login = new Promise(resolve => {

setTimeout(() => {

resolve('Login Efetuado');

}, 1000);

});

const dados = new Promise(resolve => {

setTimeout(() => {

resolve('Dados Carregados');

}, 1500);

});

const carregouPrimeiro = Promise.race([login, dados]);

carregouPrimeiro.then(resposta => {

console.log(resposta); // Login Efetuado

});

**FETCH API**

Permite fazermos requisições HTTP através do método fetch(). Este método retorna a resolução de uma Promise. Podemos então utilizar o then para interagirmos com a resposta, que é um objeto do tipo Response.

fetch('./arquivo.txt').then(function(response) {

console.log(response); // Response HTTP (Objeto)

});

**RESPONSE**

O objeto Response, possui um corpo com o conteúdo da resposta. Esse corpo pode ser transformado utilizando métodos do protótipo do objeto Response. Estes retornam outras promises.

fetch('./arquivo.txt').then(function(response) {

return response.text();

}).then(function(corpo) {

console.log(corpo);

});

**.JSON()**

Um tipo de formato de dados muito utilizado com JavaScript é o JSON (JavaScript Object Notation), pelo fato dele possuir basicamente a mesma sintaxe que a de um objeto js. .json() transforma um corpo em json em um objeto JavaScript.

fetch('https://viacep.com.br/ws/01001000/json/')

.then(response => response.json())

.then(cep => {

console.log(cep.bairro, cep.logradouro);

});

**.TEXT()**

Podemos utilizar o .text() para diferentes formatos como txt, json, html, css, js e mais.

const styleElement = document.createElement('style');

fetch('./style.css')

.then(response => response.text())

.then(style => {

styleElement.innerHTML = style;

document.body.appendChild(styleElement);

});

**.BLOB()**

Um blob é um tipo de objeto utilizado para representação de dados de um arquivo. O blob pode ser utilizado para transformarmos requisições de imagens por exemplo. O blob gera um URL único.

const div = document.createElement('div');

fetch('./imagem.png')

.then(response => response.blob())

.then(imgBlob => {

const blobUrl = **URL.createObjectURL(imgBlob);**

console.log(blobUrl);

});

**.CLONE()**

Ao utilizarmos os métodos acima, text, json e blob, a resposta é modificada. Por isso existe o método clone, caso você necessite transformar uma resposta em diferentes valores.

const div = document.createElement('div');

fetch('https://viacep.com.br/ws/01001000/json/')

.then(response => {

const cloneResponse = response.clone();

response.json().then(json => {

console.log(json)

});

cloneResponse.text().then(text => {

console.log(text)

});

});

**.HEADERS**

É uma propriedade que possui os cabeçalhos da requisição. É um tipo de dado iterável então podemos utilizar o forEach para vermos cada um deles.

const div = document.createElement('div');

fetch('https://viacep.com.br/ws/01001000/json/')

.then(response => {

response.headers.forEach(console.log);

});

**.STATUS E .OK**

Retorna o status da requisição. Se foi 404, 200, 202 e mais. ok retorna um valor booleano sendo true para uma requisição de sucesso e false para uma sem sucesso.

const div = document.createElement('div');

fetch('https://viacep.com.br/ws/01001000/json/')

.then(response => {

console.log(response.status, response.ok);

if(response.status === 404) {

console.log('Página não encontrada')

}

});

**.URL E .TYPE**

.url retorna o url da requisição. .type retorna o tipo da reposta.

const div = document.createElement('div');

fetch('https://viacep.com.br/ws/01001000/json/')

.then(response => {

console.log(response.type, response.url);

});

//types

// basic: feito na mesma origem

// cors: feito em url body pode estar disponível

// error: erro de conexão

// opaque: no-cors, não permite acesso de outros sites

**Como buscar por elementos dentro de um arquivo HTML:**

const sobre = fetch('./sobre.html');

const div = document.createElement('div');

sobre.then(r => r.text())

.then(body => {

div.innerHTML = body;

const titulo = div.querySelector('h1');

console.log(titulo);

})

**JSON.PARSE() E JSON.STRINGIFY()**

JSON.parse() irá transformar um texto JSON em um objeto JavaScript. JSON.stringify() irá transformar um objeto JavaScript em uma string no formato JSON.

const textoJSON = '{"id": 1, "titulo": "JavaScript", "tempo": "25min"}';

const textoOBJ = JSON.parse(textoJSON);

const enderecoOBJ = {

cidade: 'Rio de Janeiro',

rua: 'Ali Perto',

pais: 'Brasil',

numero: 50,

};

const enderecoJSON = JSON.stringify(enderecoOBJ);

**URL E METHOD**

Uma requisição HTTP é feita através de uma URL. O método padrão é o GET, mas existem outros como POST, UPDATE, DELETE, HEAD.

const url = 'https://jsonplaceholder.typicode.com/posts/';

**const options = {**

**method: 'POST',**

**headers: {**

**"Content-Type": "application/json; charset=utf-8",**

**},**

**body: '"aula": "JavaScript"',**

**}**

fetch(url, options);

.then(response => response.json())

.then(json => {

console.log(json);

});

**CORS**

Cross-Origin Resource Sharing, gerencia como deve ser o compartilhamento de recursos entre diferente origens.

É definido no servidor se é permitido ou não o acesso dos recursos através de scripts por outros sites. Utilizando o Access-Control-Allow-Origin.

Se o servidor não permitir o acesso, este será bloqueado. É possível passar por cima do bloqueio utilizando um proxy.

CORS é um acordo entre browser / servidor ou servidor / servidor. Ele serve para dar certa proteção ao browser, mas não é inviolável.

const url = 'https://cors-anywhere.herokuapp.com/https://www.google.com/';

const div = document.createElement('div');

fetch(url)

.then(r => r.text())

.then(r => {

div.innerHTML = r;

console.log(div);

});

**ASYNC / AWAIT**

A palavra chave async indica que a função possui partes assíncronas e que você pretende esperar a resolução da mesma antes de continuar. O await irá indicar a promise que devemos esperar. Faz parte do ES8.

async function puxarDados() {

const dadosResponse = await fetch('./dados.json');

const dadosJSON = await dadosResponse.json();

document.body.innerText = dadosJSON.titulo;

}

puxarDados();

**TRY / CATCH**

Para lidarmos com erros nas promises, podemos utilizar o try e o catch na função.

async function puxarDados() {

try {

const dadosResponse = await fetch('./dados.json');

const dadosJSON = await dadosResponse.json();

document.body.innerText = dadosJSON.titulo;

}

catch(erro) {

console.log(erro);

}

}

puxarDados();

**INICIAR FETCH AO MESMO TEMPO**

Não precisamos esperar um fetch para começarmos outro. Porém precisamos esperar a resposta resolvida do fetch para transformarmos a response em json.

async function iniciarAsync() {

const dadosResponse = fetch('./dados.json');

const clientesResponse = fetch('./clientes.json');

// ele espera o que está dentro da expressão () ocorrer primeiro

const dadosJSON = await (**await dadosResponse**).json();

const clientesJSON = await (**await clientesResponse**).json();

}

iniciarAsync();

**HISTORY**

É possível acessarmos o histórico de acesso do browser em uma sessão específica através do window.history. O objeto history possui diferentes métodos e propriedades.

window.history;

window.history.back(); // vai para a anterior

window.history.forward(); // vai para a próxima

**PUSHSTATE()**

A parte interessante de manipularmos o history é que podemos modificar o histórico e adicionar um novo item. window.history.pushState(obj, title, url).

// Em obj podemos enviar um objeto com dados

// mas o seu uso é restrito por isso geralmente utilizamos

// null. O title ainda é ignorado por alguns browsers, também

// utilizamos null nele. O url que é parte importante.

window.history.pushState(null, null, 'sobre.html');

**POPSTATE**

O evento popstate pode ser adicionado ao objeto window. Assim podemos executar uma função toda vez que o usuário clicar no botão de voltar ou próximo.

window.addEventListener('popstate', () => {

fetchPage(window.location.pathname);

});

**FETCH E HISTORY**

Ao puxarmos dados via fetch api, o url da página continua o mesmo. Ao combinar fetch com a history api conseguimos simular uma navegação real entre páginas, sem a necessidade de recarregamento da mesma.

async function fetchPage(url) {

const pageReponse = await fetch(url);

const pageText = await pageReponse.text();

window.history.pushState(null, null, url);

}

**Classes**

O ES6 trouxe uma nova sintaxe para implementarmos funções construtoras. Agora podemos utilizar a palavra chave class. É considerada syntactical sugar, pois por baixo dos panos continua utilizado o sistema de protótipos de uma função construtora para criar a classe.

class Button {

constructor(text, background) {

this.text = text;

this.background = background;

}

element() {

const buttonElement = document.createElement('button');

buttonElement.innerText = this.text;

buttonElement.style.background = this.background;

return buttonElement;

}

}

const blueButton = new Button('Comprar', 'blue');

**Utilizando constructor function, ficaria assim:**

function Button(text, background) {

this.text = text;

this.background = background;

}

Button.prototype.element = function() {

const buttonElement = document.createElement('button');

buttonElement.innerText = this.text;

buttonElement.style.background = this.background;

return buttonElement;

}

const blueButton = new Button('Comprar', 'blue');

**STATIC VS PROTOTYPE**

Por padrão todos os métodos criados dentro da classe irão para o protótipo da mesma. Porém podemos criar métodos diretamente na classe utilizando a palavra chave static. Assim como [].map() é um método de uma array e Array.from() é um método do construtor Array.

class Button {

constructor(text) {

this.text = text;

}

static create(background) {

const elementButton = document.createElement('button');

elementButton.style.background = background;

elementButton.innerText = 'Clique';

return elementButton;

}

}

const blueButton = Button.create('blue');

**USO COMUM DO STATIC**

Você pode utilizar um método static para retornar a própria classe com propriedades já pré definidas.

class Button {

constructor(text, background) {

this.text = text;

this.background = background;

}

element() {

const elementButton = document.createElement('button');

elementButton.innerText = this.text;

elementButton.style.background = this.background;

return elementButton

}

static createBlue(text) {

return new Button(text, 'blue');

}

}

const blueButton = Button.createBlue('Comprar');

**GET E SET**

Podemos definir comportamentos diferentes de get e set para um método.

const button = {

**get element()** {

return this.\_element;

}

**set element(tipo)** {

this.\_element = document.createElement(tipo);

}

}

button.element = 'button'; // set

button.element; // get (<button></button>);

**VALOR ESTÁTICO**

Se definirmos apenas o get de um método, teremos então um valor estático que não será possível mudarmos.

const matematica = {

get PI() {

return 3.14;

},

};

matematica.PI; // get (3.14)

matematica.PI = 20; // nada acontece

**SUBCLASSES**

É possível criarmos uma subclasse, esta irá ter acesso aos métodos da classe à qual ela estendeu através do seu protótipo.

class Veiculo {

constructor(rodas) {

this.rodas = rodas;

}

acelerar() {

console.log('Acelerou');

}

}

class Moto extends Veiculo {

empinar() {

console.log('Empinou com ' + this.rodas + ' rodas');

}

}

const honda = new Moto(2);

honda.empinar();

**SUPER**

É possível utilizar a palavra chave super para falarmos com a classe que pai e acessarmos os seus métodos e propriedades. Caso o construtor seja sobrescrito, é obrigatório chamar um super(roda, capacete) dentro do novo construtor para que o construtor do pai seja acionado.

class Veiculo {

constructor(rodas) {

this.rodas = rodas;

}

acelerar() {

console.log('Acelerou');

}

}

class Moto extends Veiculo {

acelerar() {

super.acelerar();

console.log('Muito');

}

}

const honda = new Moto(2);

honda.acelerar();

**FUNCTION DECLARATION**

São duas as formas mais comuns de declararmos uma função. A que utilizamos até o momento é chamado de Function Declaration.

function somar(a,b) {

return a + b;

}

somar(4,2); // 6

**FUNCTION EXPRESSION**

É criada a partir da declaração de uma variável, na qual assinalamos uma função. Esta função pode ser anônima ou nomeada. A mesma poderá ser ativada através da variável assinalada.

const somar = function(a,b) {

return a + b;

}

somar(4,2); // 6

**Vantagens da function expression:**

* Não são declaradas no hosisting, fazendo com que as funções tenham de ser declarada na ordem em que serão utilizadas (caso uma função tente utilizar uma que está abaixo dela, dará erro, pois a variável será undefined). Isso deixa o código mais estruturado;
* As function declaration se tornam métodos do objeto window, vazando o escopo de bloco e podendo sobrescrever funções do objeto window, já as function expression, não.

**IIFE - IMMEDIATELY INVOKED FUNCTION EXPRESSION**

Antes da introdução de modules e da implementação do escopo de bloco, a forma mais comum utilizada para isolarmos o escopo de um código JavaScript era através das chamadas IIFE's (a função já é criada e ativada de uma vez).

var instrumento = 'Violão';

**(function() {**

// código isolado do escopo global

var instrumento = 'Guitarra';

console.log(instrumento); // Guitarra

**})();**

console.log(instrumento); // Violão

**IIFE - ARROW FUNCTION**

Compiladores ainda transformam modules em IIFE's para manter a compatibilidade com browsers antigos.

const instrumento = 'Violão';

**(() => {**

const instrumento = 'Guitarra';

console.log(instrumento); // Guitarra

**})();**

console.log(instrumento); // Violão

**FACTORY FUNCTION**

São funções que retornam um objeto **sem a necessidade de utilizarmos a palavra chave new**. Possuem basicamente a mesma função que constructor functions / classes.

function createButton(text) {

function element() {

const buttonElement = document.createElement('button');

buttonElement.innerText = text;

return buttonElement;

}

**return {**

**element: element,**

**text: text,**

**}**

}

const comprarBtn = createButton('Comprar');

Podemos ter métodos e atributos privados, bastando não incluí-los no objeto retornado.

**ICE FACTORY**

Podemos impedir que os métodos e propriedades sejam modificados com Object.freeze().

return **Object.freeze(**{

nome,

sobrenome,

andar,

});

const designer = criarPessoa('André', 'Rafael');

**CONSTRUCTOR FUNCTION / FACTORY FUNCTION**

Uma das vantagens da Factory Function é a possibilidade de iniciarmos a mesma sem a utilização da palavra chave new. Também é possível fazer isso com uma Constructor Function.

function Pessoa(nome) {

if (!(this instanceof Pessoa)) // ou (!new.target)

return new Pessoa(nome);

this.nome = nome;

}

Pessoa.prototype.andar = function() {

return `${this.nome} andou`;

}

const designer = Pessoa('André');

(Quando se utiliza o new, o this referencia uma instância de Pessoa, caso não utilize, ele referencia uma instância de Window, daí a checagem para recriar utilizando o new se necessário)

**EXEMPLO REAL (Parecido com o JQuery)**

function $$(selectedElements) {

const elements = document.querySelectorAll(selectedElements);

function on(onEvent, callback) {

elements.forEach(element => {

element.addEventListener(onEvent, callback);

});

return this; **//retorna o this para que seja possível encadear vários comandos**

}

function hide() {

elements.forEach(element => {

element.style.display = 'none';

});

return this;

}

function show() {

elements.forEach(element => {

element.style.display = 'initial';

});

return this;

}

return Object.freeze({

elements,

on,

hide,

show,

});

}

const buttons = $$('button');

buttons.hide().show().addClass('ativo').removeClass('ativo');

**CLOSURES**

A funcao2 possui 4 escopos. O primeiro escopo é o Local, com acesso ao item3. O segundo escopo dá acesso ao item2, esse escopo é chamado de Clojure (funcao1) (escopo de função dentro de função). O terceiro escopo é o Script com acesso ao item1 e o quarto escopo é o Global/Window.

let item1 = 1;

function funcao1() {

let item2 = 2;

function funcao2() {

let item3 = 3;

console.log(item1);

console.log(item2);

console.log(item3);

}

funcao2();

}

**EXEMPLO DO USO DE CLOSURES**

Um dos casos mais clássicos para a demonstração de closures é através da criação de uma função de incremento. É como se a função incrementar carregasse uma mochila chamada contagem, onde uma referência para as suas variáveis estão contidas na mesma.

function contagem() {

let total = 0;

return function incrementar() {

total++;

console.log(total);

}

}

const ativarIncrementar = contagem();

ativarIncrementar(); // 1

ativarIncrementar(); // 2

ativarIncrementar(); // 3

**DESTRUCTURING**

Permite a desestruturação de Arrays e Objetos. Atribuindo suas propriedades à novas variáveis.

const carro = {

marca: 'Fiat',

ano: 2018,

portas: 4,

}

const {marca, ano} = carro;

console.log(marca); // Fiat

console.log(ano); // 2018

**NESTING**

É possível aninhar uma desestruturação dentro de outra.

const cliente = {

nome: 'Andre',

compras: {

digitais: {

livros: ['Livro 1', 'Livro 2'],

videos: ['Video JS', 'Video HTML']

},

fisicas: {

cadernos: ['Caderno 1']

}

}

}

const {fisicas, digitais, digitais: {livros, videos}} = cliente.compras;

console.log(fisicas);

console.log(livros);

console.log(videos);

console.log(digitais);

**NOME DAS VARIÁVEIS**

É necessário indicar o nome da propriedade que você deseja desestruturar de um objeto. É possível mudar o nome da variável final com:

const cliente = {

nome: 'Andre',

compras: 10,

}

const {nome, compras} = cliente;

// ou

const {nome: nomeCliente, compras: comprasCliente} = cliente;

**VALOR INICIAL**

Caso a propriedade não exista o valor padrão dela será undefined. É possível modificar este valor no momento da desestruturação.

const cliente = {

nome: 'Andre',

compras: 10,

}

const {nome, compras, email = 'email@gmail.com', cpf} = cliente;

console.log(email) // email@gmail.com

console.log(cpf) // undefined

**ARGUMENTO DESESTRUTURADO**

Se uma função espera receber como argumento um objeto, podemos desestruturar ele no momento da declaração.

function handleKeyboard(event) {

console.log(event.key);

}

// Com Destructuring

function handleKeyboard({key}) {

console.log(key);

}

document.addEventListener('keyup', handleKeyboard);

**STYLE VS GETCOMPUTEDSTYLE**

const btn = document.querySelector('button');  
console.log(btn.style); //Serve apenas para eu definir valores, estando todo em branco inicialmente, apenas o que eu definir via JS aparecerá

console.log(getComputedStyle(btn)); //Mostra como o botão está atualmente

**ARGUMENTS**

A palavra chave arguments é um objeto Array-like criado dentro da função. Esse objeto contém os valores dos argumentos.

function perimetroForma(lado, totalLados = 4) {

const argArray = Array.from(arguments);

}

**PARÂMETRO REST**

É possível declararmos um parâmetro utilizando ... na frente do mesmo. Assim todos os argumentos que passarmos na ativação da função, ficarão dentro do parâmetro.

function anunciarGanhadores(...ganhadores) {

ganhadores.forEach(ganhador => {

console.log(ganhador + ' ganhou.')

});

}

anunciarGanhadores('Pedro', 'Marta', 'Maria');

**OPERADOR SPREAD**

Assim como o rest, o operador Spread também utiliza os ... para ser ativado. O spread irá distribuir um item iterável, um por um.

const frutas = ['Banana', 'Uva', 'Morango'];

const legumes = ['Cenoura', 'Batata'];

const comidas = [...frutas, 'Pizza', ...legumes];

**SPREAD ARGUMENT**

O Spread pode ser muito útil para funções que recebem uma lista de argumentos ao invés de uma array.

const numeroMaximo = Math.max(4,5,20,10,30,2,33,5); // 33

const listaNumeros = [1,13,21,12,55,2,3,43];

const numeroMaximoSpread = Math.max(...listaNumeros);

**TRANSFORMAR EM ARRAY**

É possível transformar itens iteráveis em uma array real com o spread.

const btns = document.querySelectorAll('button');

const btnsArray = [...btns];

const frase = 'Isso é JavaScript';

const fraseArray = [...frase];

**ITERABLE**

São objetos que possuem o método [Symbol.iterator], geralmente no protótipo, é dentro dele que a função que lida com a iteração será definida. Ex: Array, String, NodeList, boa parte das Array-Like e outros.

const frutas = ['Banana', 'Morango', 'Uva'];

const frase = 'Isso é JavaScript';

fetch('https://pokeapi.co/api/v2/pokemon/')

.then(({headers}) => console.log(headers));

**FOR...OF**

É possível fazemos um loop por cada iteração do objeto iterável utilizando o for...of. Além deste loop podemos também utilizar o Spread Operator nos mesmos.

const frutas = ['Banana', 'Morango', 'Uva'];

const frase = 'Isso é JavaScript';

for(const fruta of frutas) {

console.log(fruta);

}

for(const char of frase) {

console.log(char);

}

**SPREAD E FOR...OF**

Com o for loop podemos manipular cada um dos elementos do objeto iterável.

const buttons = document.querySelectorAll('button');

for(const btn of buttons) {

btn.style.background = 'blue';

}

console.log(...buttons);

**FOR...IN**

Este loop irá retornar a chave (key) de todas as propriedades enumeráveis (que não sejam símbolos) de um objeto.

const carro = {

marca: 'Honda',

ano: 2018,

}

for(const propriedade in carro) {

console.log(propriedade);

}

**DO / WHILE**

Outro tipo de loop é o Do / While. Não é muito utilizado.

let i = 0;

do {

console.log(i++)

} while (i <= 5);

**REGEXR**

Site para testar expressões regulares: https://regexr.com/

**REGULAR EXPRESSION**

Regexp ou Regex são expressões utilizadas para realizarmos buscas / substituições de padrões em strings. Os padrões devem ser colocados entre //. Geralmente vamos utilizá-las nos métodos .replace() e .split().

// Procura: a

const padraoRegexp = /a/;

const texto = 'JavaScript';

const novoTexto = texto.replace(padraoRegexp, 'B');

// BavaScript

**LITERAL**

Utilizar um caracter literal irá realizar uma busca específica deste caracter.

// Procura: J seguido de a, v e a

const regexp = /Java/;

'JavaScript'.replace(regexp, 'Type');

// TypeScript

**FLAG: G**

As flags irão modificar como a expressão é interpretada. Uma das mais utilizadas é a **g**, que significa global, ou seja, retorne todos os resultados que estiverem dentro do padrão e não apenas o primeiro. A flag deve ser colocada no final da expressão.

// Procura: Todo a

const regexp = /a/g;

'JavaScript'.replace(regexp, 'i');

// JiviScript

**FLAG: I**

Com o **i** informamos que devem ser ignoradas as diferenças entre maiúsculas e minúsculas. Isso significa que /a/ irá buscar por a e A.

// Procura: Todo PE, Pe, pE e pe

const regexp = /Pe/gi;

'Perdeu perdido'.replace(regexp, 'Ba');

// Bardeu Bardido

**CHARACTER CLASS**

Se colocarmos os caracteres entre colchetes **[]**, estamos definindo uma classe. /[ab]/ irá procurar por a **ou** por b.

// Procura: Todo a, A, i, I

const regexp = /[ai]/gi;

'JavaScript'.replace(regexp, 'u');

// JuvuScrupt

**UM OU OUTRO**

Combine caracteres literais com uma classe para buscarmos variações: Ju[nl]ho busca Julho ou Junho.

// Procura: B, seguido de r, a

// seguido de s ou z, seguido de i, l

const regexp = /Bra[sz]il/g;

'Brasil é com z: Brazil'.replace(regexp, 'Prazer');

// Prazer é com z: Prazer

**DE A À Z**

O traço **-** dentro de [] pode servir para definirmos um alcance. [A-Z] irá buscar os caracteres de A à Z. [0-9] busca de 0 à 9. A tabela UNICODE é utilizada como referência para definir os caracteres dentro do alcance.

// Busca por itens de a à z

const regexp = /[a-z]/g;

'JavaScript é a linguagem.'.replace(regexp, '0');

// J000S00000 é 0 000000000.

// Busca por itens de a à z e A à Z

const regexp = /[a-zA-Z]/g;

'JavaScript é a linguagem.'.replace(regexp, '1');

// 1111111111 é 1 111111111.

// Busca por números de 0 à 9

const regexpNumero = /[0-9]/g;

'123.333.333-33'.replace(regexpNumero, 'X');

// XXX.XXX.XXX-XX

**NEGAR**

Utilizando o acento circunflexo **^** podemos negar caracteres. Ou seja, pegue tudo que não seja [^a]

// Procura: tudo que não estiver entre a e z

const regexp = /[^a-z]/g;

'Brasil é com z: Brazil'.replace(regexp, ' ');

// rasil com z razil

**PONTO**

O ponto **.** irá selecionar qualquer caracter, menos quebras de linha.

// Procura: todos os caracteres menos quebra de linha

const regexp = /./g;

'JavaScript é a linguagem.'.replace(regexp, '0');

// 0000000000000000000000000

**ESCAPAR ESPECIAIS**

Caracteres especiais como o ponto ., podem ser escapados utilizando a barra **\**, assim este não terá mais a sua função especial e será tratado como literal. **Lista de caracteres especiais: +\*?^$\.[]{}()|/**

// Procura: todos os pontos

const regexp = /\./g;

const regexpAlternativa = /[.]/g;

'999.222.222.11'.replace(regexp, '-');

// 999-222-222-11

**WORD**

O **\w** irá selecionar qualquer caracter alfanumérico e o underline. É a mesma coisa que [A-Za-z0-9\_].

// Procura: todos os alfanuméricos

const regexp = /\w/g;

'Guarda-chuva R$ 23,00.'.replace(regexp, '-');

// ------------ -$ --,--.

**NOT WORD**

O **\W** irá selecionar tudo o que não for caracter alfanumérico e o underline. É a mesma coisa que [^A-Za-z0-9\_].

// Procura: o que não for caracter alfanuméricos

const regexp = /\W/g;

'Guarda-chuva R$ 23,00.'.replace(regexp, '-');

// Guarda-chuva-R--23-00-

**DIGIT**

O **\d** irá selecionar qualquer dígito. É a mesma coisa que [0-9].

// Procura: todos os dígitos

const regexp = /\d/g;

'+55 (21) 2222-2222'.replace(regexp, 'X');

// +XX (XX) XXXX-XXXX.

**NOT DIGIT**

O **\D** irá selecionar tudo que não for dígito. É a mesma coisa que [^0-9].

// Procura: o que não for dígito

const regexp = /\D/g;

'+55 (21) 2222-2222'.replace(regexp, '');

// 552122222222

**WHITESPACE**

O \s irá selecionar qualquer espaço em branco, isso inclui espaços, tabs, quebra de linhas.

// Procura: espaços em branco

const regexp = /\s/g;

'+55 (21) 2222- 2222 '.replace(regexp, '');

// +55(21)2222-2222

**NOT WHITESPACE**

O \S irá selecionar qualquer caracter que não for espaço em branco.

// Procura: o que não for espaço em branco

const regexp = /\S/g;

'+55 (21) 2222- 2222 '.replace(regexp, '');

// XXX XXXX XXXXX XXXX

/[\s\S]/g irá selecionar tudo.

**QUANTIFICADOR**

É possível selecionar caracteres seguidos, como /bbb/g irá selecionar apenas bbb. Com as chaves **{}** podemos indicar a repetição /b{3}/g. Agora ele está fazendo uma seleção completa e não caracter por caracter.

// Procura: 4 a's seguidos

const regexp = /aaaa/g;

const regexpAlt = /a{4}/g;

'Vaaaai ali por favor.'.replace(regexp, 'a');

// Vai ali por favor.

**QUANTIFICADOR MIN E MAX**

Podemos informar o min e max do quantificador /a{2,4}/ vai selecionar quando aparecer a duas vezes ou até 4 vezes. /a{2,}/ irá selecionar quando se repetir duas ou mais vezes.

// Procura: dígitos seguidos de 2 à 3

const regexp = /\d{2,3}/g;

'222.333.222.42'.replace(regexp, 'X');

// X.X.X.X

// Procura: letras seguidos com 1 caracter ou mais

const regexpLetras = /\w{1,}/g;

'A melhor linguagem é JavaScript'.replace(regexpLetras, 'X');

// X X X é X

**MAIS +**

O sinal de + significa que devemos selecionar quando existir pelo menos uma ou mais ocorrências.

// Procura: dígitos em ocorrência de um ou mais

const regexp = /\d+/g;

'222.333.222.42'.replace(regexp, 'X');

// X.X.X.X

// Procura: Começa com d, seguido por uma ou mais letras.

const regexpLetras = /d\w+/g;

'Dígitos, dados, desenhos, Dito, d'.replace(regexpLetras, 'X');

// Dígitos, X, X, Dito, d

**ASTERISCO \***

O sinal \* significa que devemos selecionar quando existir 0 ou mais ocorrências.

// Procura: Começa com d, seguido por zero ou mais letras.

const regexp = /d\w\*/g;

'Dígitos, dados, desenhos, Dito, d'.replace(regexp, 'X');

// Dígitos, X, X, Dito, X

**OPCIONAL ?**

O sinal ? significa que o caracter é opcional, pode ou não existir.

// Procura: Por regex com p opcional

const regexp = /regexp?/g;

'Qual é o certo, regexp ou regex?'.replace(regexp, 'Regular Expression');

// Qual é o certo, Regular Expression ou Regular Expression?

**ALTERNADO |**

O sinal | irá selecionar um ou outro. java|php

// Procura: java ou php (case insensitive)

const regexp = /java|php/gi;

'PHP e Java são linguagens diferentes'.replace(regexp, 'X');

// X e X são linguagens diferente

**WORD BOUNDARY \b**

O sinal **\b** irá indicar que pretendemos fazer uma seleção que **deve ter início e fim de não caracteres** \w.

// Procura: java (case insensitive)

const regexp = /java/gi;

'Java não é JavaScript.'.replace(regexp, 'X');

// X não é XScript.

// Procura: java (case insensitive)

const regexpBoundary = /\bjava\b/gi;

'Java não é JavaScript.'.replace(regexpBoundary, 'X');

// X não é JavaScript.

// Procura: Dígitos em sequência, que estejam isolados

const regexpDigito = /\b\d+\b/gi;

'O Restaurante25 na Rua 3, custa R$ 32,00'.replace(regexDigito, 'X');

// O Restaurante25 na Rua X, custa R$ X,X

'11\_22 33-44 55é66 77e88'.replace(regexpDigito, 'X');

// 11\_22 X-X XéX 77e88

**NOT WORD BOUNDARY \B**

É o contrário do \b.

const regexpDigito = /\B\d+\B/gi;

'11\_22 33-44 55é66 77e88'.replace(regexpDigito, 'X');

// 1X\_X2 33-44 55é66 7XeX8

**ANCHOR BEGINNING**

Com o ^ é possível informar que a busca deve ser iniciada no início da linha.

// Procura: sequência de alfanuméricos

// no início da linha.

const regexp = /^\w+/g;

`andre@origamid.com

contato@origamid.com`.replace(regexp, 'X');

// X@origamid.com

// contato@origamid.com

**ANCHOR END**

Com o $ é possível informar que a busca deve ser iniciada no final da linha.

// Procura: sequência de alfanuméricos

// no final da linha.

const regexp = /\w+$/g;

`andre@origamid.com

contato@origamid.com`.replace(regexp, 'X');

// andre@origamid.com

// contato@origamid.X

**FLAG: M**

Com a flag **m** de multiline, podemos informar que a busca de início ^ e final $ de linha devem ocorrer em todas as linhas disponíveis.

// Procura: sequência de alfanuméricos

// no final da linha.

const regexp = /\w+$/gm;

`andre@origamid.com

contato@origamid.com`.replace(regexp, 'X');

// andre@origamid.X

// contato@origamid.X

// Procura: sequência de alfanuméricos

// no início da linha.

const regexp = /^\w+/gm;

`andre@origamid.com

contato@origamid.com`.replace(regexp, 'X');

// X@origamid.com

// X@origamid.com

**LINE FEED \n**

O \n irá selecionar o final de uma linha, quando criamos uma nova.

const regexp = /\n/g;

`andre@origamid.com\ncontato@origamid.com`.replace(regexp, '---');

// andre@origamid.com---contato@origamid.com

`andre@origamid.com

contato@origamid.com`.replace(regexp, 'X');

// andre@origamid.com---contato@origamid.com

\t seleciona tabs

**UNICODE \U**

O \u irá selecionar o respectivo caracter unicode, de acordo com o código passado em \uXXXX. Ex: \u0040 seleciona o @.

// Procura: @ ou ©

const regexp = /\u0040|\u00A9/g;

'andre@origamid.com ©'.replace(regexp, '---');

// andre---origamid.com ---

**REFERÊNCIA DA SELEÇÃO**

É possível utilizarmos o $& durante o momento da substituição para fazermos uma referência à seleção.

// Procura: Java

const regexp = /Java/g;

'PHP e Java são linguagens diferentes'.replace(regexp, '--$&Script');

// PHP e --JavaScript são linguagens diferentes

// $& será igual à Java

**GRUPO DE CAPTURA**

É possível definirmos diferentes grupos de captura, que poderão ser referenciados durante a substituição. Basta envolvermos um grupo entre () parênteses. A referência se cada grupo será feita com $n, sendo o primeiro $1.

// Procura: sequência alfanumérica, seguida

// de @, seguido de alfanumérico ou .

const regexp = /(\w+)@[\w.]+/g;

'andre@email.com.br'.replace(regexp, '$1@gmail.com');

// andre@gmail.com

**MAIS DE UM GRUPO**

Podemos definir quantos grupos de captura quisermos.

// Procura: sequência alfanumérica, seguida

// de , seguido espaço de sequência alfanumérica.

const regexp = /(\w+),\s(\w+)/g;

'Rafael, Andre'.replace(regexp, '$2 $1');

// Andre Rafael

**MAIS DO QUE CAPTURA APENAS**

Um grupo também serve para agruparmos uma sequência de caracteres que queremos em repetição.

// Procura: qualquer sequência de ta

const regexp = /(ta)+/gi;

'Tatata, tata, ta'.replace(regexp, 'Pá');

// Pá, Pá, Pá

**IGNORAR CAPTURA**

Utilize o (?:) para ignorar a captura. (O grupo irá valer, mas o valor não será capturado)

// Procura: qualquer sequência de ta

const regexp = /(?:ta)+/gi;

'Tatata, tata, ta'.replace(regexp, 'Pá');

// Pá, Pá, Pá

**POSITIVE LOOKAHEAD**

Faz a seleção dos itens que possuírem o padrão dentro de (?=) à sua frente. Apesar de utilizar () parênteses o positive lookahead não captura grupo.

// Procura: dígitos em sequência, que

// possuírem px, sem selecionar o px.

const regexp = /\d(?=px)/g;

'2em, 4px, 5%, 2px, 1px'.replace(regexp, 'X');

// 2em, Xpx, 5%, Xpx, Xpx

**NEGATIVE LOOKAHEAD**

Faz a seleção dos itens não possuírem o padrão dentro de (?!) à sua frente.

// Procura: dígitos que não possuírem px

// sem selecionar o restante.

const regexp = /\d(?!px)/g;

'2em, 4px, 5%, 5px, 1px'.replace(regexp, 'X');

// Xem, 4px, X%, 5px, 1px

**POSITIVE LOOKBEHIND**

Faz a seleção dos itens que possuírem o padrão dentro de (?<=) atrás dos mesmos.

// Procura: dígitos que possuírem R$

// na frente dos mesmos

const regexp = /(?<=R\$)[\d]+/g;

'R$99, 100, 200, R$20'.replace(regexp, 'X');

// R$X, 100, 200, R$X

**CEP**

const regexpCEP = /\d{5}[-\s]?\d{3}/g;

const ceps = ['00000-000', '00000 000', '00000000'];

**CPF**

const regexpCPF = /(?:\d{3}[-.]?){3}\d{2}/g;

const cpfs = ['000.000.000-00', '000-000-000-00', '000.000.000.00', '000000000-00', '00000000000'];

**CNPJ**

const regexpCNPJ = /\d{2}[-.]?(?:\d{3}[-.]?){2}[-\/]?\d{4}[-.]?\d{2}/g;

const cnpjs = [ '00.000.000/0000-00', '00000000000000', '00-000-000-0000-00', '00.000.000/000000', '00.000.000.000000', '00.000.000.0000.00',];

**TELEFONE**

const regexpTELEFONE = /(?:\+?55\s?)?(?:\(?\d{2}\)?[-\s]?)?\d{4,5}[-\s]?\d{4}/g;

const telefones = ['+55 11 98888-8888', '+55 11 98888 8888', '+55 11 988888888',

'+55 11988888888', '+5511988888888', '5511988888888', '11 98888-8888', '11 98888 8888', '(11) 98888 8888', '(11) 98888-8888', '11-98888-8888', '11 98888 8888', '11988888888', '11988888888', '988888888', '(11)988888888', '98888 8888', '8888 8888'];

**EMAIL**

const regexpEMAIL = /[\w.-]+@[\w-]+\.[\w-.]+/gi;

const emails = ['email@email.com', 'email@email.com.org', 'email-email@email.com', 'email\_email@email.com', 'email.email23@email.com.br', 'email.email23@empresa-sua.com.br', 'c@contato.cc',];

**TAG**

const regexpTAG = /<\/?[\w\s="']+\/?>/gi;

const tags = [

'<div>Isso é uma div</div>',

'<div class="ativa">Essa está ativa</div>',

'<img src="imagem" />',

'<img src="imagem">',

'<ul class="ativa">',

'<li>Essa está ativa</li>',

'</ul>'

];

**TAG APENAS O NOME**

const regexpTAG = /(?<=<\/?)[\w]+/gi;

Esta parte “(?<=<)”se chama positive lookbehind e quer dizer que deve começar com o caracter <.

const tags = [

'<div>Isso é uma div</div>',

'<div class="ativa">Essa está ativa</div>',

'<img src="imagem" />',

'<img src="imagem">',

'<ul class="ativa">',

'<li>Essa está ativa</li>',

'</ul>'

];

**REGEXP CONSTRUCTOR**

Toda regexp é criada com o constructor RegExp() e herda as suas propriedades e métodos. Existem diferenças na sintaxe de uma Regexp criada diretamente em uma variável e de uma passada como argumento de RegExp.

const regexp = /\w+/gi;

// Se passarmos uma string, não precisamos dos //

// e devemos utilizar \\ para meta characters, pois é necessário

// escapar a \ especial. As Flags são o segundo argumento

const regexpObj1 = new RegExp('\\w+', 'gi');

const regexpObj2 = new RegExp(/\w+/, 'gi');

'JavaScript Linguagem 101'.replace(regexpObj1, 'X');

// X X X

// Exemplo complexo:

const regexpTELEFONE1 = /(?:\+?55\s?)?(?:\(?\d{2}\)?[-\s]?)?\d{4,5}[-\s]?\d{4}/g;

const regexpTELEFONE2 = new RegExp('(?:\\+?55\\s?)?(?:\\(?\\d{2}\\)?[-\\s]?)?\\d{4,5}[-\\s]?\\d{4}', 'g');

**PROPRIEDADES**

Uma regexp possui propriedades com informações sobre as flags e o conteúdo da mesma.

const regexp = /\w+/gi;

regexp.flags; // 'gi'

regexp.global; // true

regexp.ignoreCase; // true

regexp.multiline; // false

regexp.source; // '\w+'

**REGEXP.TEST()**

O método test() verifica se existe ou não uma ocorrência da busca. Se existir ele retorna true. A próxima vez que chamarmos o mesmo, ele irá começar do index em que parou no último true.

const regexp = /Java/g;

const frase = 'JavaScript e Java';

regexp.lastIndex; // 0

regexp.test(frase); // true

regexp.lastIndex; // 4

regexp.test(frase); // true

regexp.lastIndex; // 17

regexp.test(frase); // false

regexp.lastIndex; // 0

regexp.test(frase); // true (Reinicia

regexp.lastIndex; // 4

**TEST() EM LOOP**

Podemos utilizar o while loop, para mostrar enquanto a condição for verdadeira. Assim retornamos a quantidade de match's.

const regexp = /Script/g;

const frase = 'JavaScript, TypeScript e CoffeeScript';

let i = 1;

while(regexp.test(frase)) {

console.log(i++, regexp.lastIndex);

}

// 1 10

// 2 22

// 3 37

**REGEXP.EXEC()**

O exec() diferente do test(), irá retornar uma Array com mais informações do que apenas um valor booleano.

const regexp = /\w{2,}/g;

const frase = 'JavaScript, TypeScript e CoffeeScript';

regexp.exec(frase);

// ["JavaScript", index: 0, input: "JavaScript,

// TypeScript e CoffeeScript", groups: undefined]

regexp.exec(frase);

// ["TypeScript", index: 12, input: "JavaScript,

// TypeScript e CoffeeScript", groups: undefined]

regexp.exec(frase);

// ["CoffeeScript", index: 25, input: "JavaScript,

// TypeScript e CoffeeScript", groups: undefined]

regexp.exec(frase);

// null

regexp.exec(frase); // Reinicia

// ["JavaScript", index: 0, input: "JavaScript,

// TypeScript e CoffeeScript", groups: undefined]

**LOOP COM EXEC()**

Podemos fazer um loop com exec e parar o mesmo no momento que encontre o null.

const regexp = /\w{2,}/g;

const frase = 'JavaScript, TypeScript e CoffeeScript';

let regexpResult;

while((regexpResult = regexp.exec(frase)) !== null) {

console.log(regexpResult[0]);

}

**STR.MATCH()**

O match() é um método de strings que pode receber como argumento uma Regexp. Existe uma diferença de resultado quando utilizamos a flag g ou não.

const regexp = /\w{2,}/g;

const regexpSemG = /\w{2,}/;

const frase = 'JavaScript, TypeScript e CoffeeScript';

frase.match(regexp);

// ['JavaScript', 'TypeScript', 'CoffeeScript']

frase.match(regexpSemG);

// ["JavaScript", index: 0, input: "JavaScript,

// TypeScript e CoffeeScript", groups: undefined]

Se não tiver match retorna null

**STR.SPLIT()**

O split serve para distribuirmos uma string em uma array, quebrando a string no argumento que for passado. Este método irá remover o match da array final.

const frase = 'JavaScript, TypeScript, CoffeeScript';

frase.split(', ');

// ["JavaScript", "TypeScript", "CoffeeScript"]

frase.split(/Script/g);

// ["Java", ", Type", ", Coffee", ""]

const tags = `

<ul>

<li>Item 1</li>

<li>Item 2</li>

</ul>

`;

tags.split(/(?<=<\/?)\w+/g).join('div');

// <div>

// <div>Item 1</div>

// <div>Item 2</div>

// <div>

**STR.REPLACE()**

O método replace() é o mais interessante por permitir a **utilização de funções de callback** para cada match que ele der com a Regexp.

const tags = `

<ul>

<li>Item 1</li>

<li>Item 2</li>

</ul>

`;

tags.replace(/(?<=<\/?)\w+/g, 'div');

// <div>

// <div>Item 1</div>

// <div>Item 2</div>

// <div>

**CAPTURA**

É possível fazer uma referência ao grupo de captura dentro do argumento do replace. Então podemos utilizar $&, $1 e mais.

const tags = `

<ul>

<li>Item 1</li>

<li>Item 2</li>

</ul>

`;

tags.replace(/<li/g, '$& class="ativo"');

// <ul>

// <li class="ativo">Item 1</li>

// <li class="ativo">Item 2</li>

// </ul>

**GRUPOS DE CAPTURA**

É possível definirmos quantos grupos de captura quisermos.

const emails = `

empresa@email.com

contato@email.com

suporte@email.com

`;

emails.replace(/(\w+@)[\w.]+/g, '**$1**gmail.com');

// empresa@gmail.com

// contato@gmail.com

// suporte@gmail.com

**CALLBACK**

Para substituições mais complexas, podemos utilizar um callback como segundo argumento do replace. O valor do return será o que irá substituir cada match.

const regexp = /(\w+)(@[\w]+)/g;

const emails = `joao@homail.com.br

marta@ggmail.com.br

bruna@oulook.com.br`

emails.replace(regexp, function(...args) {

console.log(args);

if(args[2] === '@homail') {

return `${args[1]}@hotmail`;

} else if(args[2] === '@ggmail') {

return `${args[1]}@gmail`;

} else if(args[2] === '@oulook') {

return `${args[1]}@outlook`;

} else {

return 'x';

}

});

// joao@hotmail.com.br

// marta@gmail.com.br

// bruna@outlook.com.br

**NPM (NODE PACKAGE MANAGER)**

É um gerenciador de pacotes feito para Node.js

Para iniciar um projeto:

npm init

Para instalar um pacote chamado Lodash:

npm install lodash

Para instalar os pacotes caso já exista um arquivo package.json:

npm install

Para instalar um pacote globalmente, basta adicionar o parâmetro -g:

npm install eslint -g

Para atualizar o pacote lodash:

npm update lodash

Para desinstalar o pacote lodash:

npm uninstall lodash

**ESLINT**

Serve para evitar problemas e padronizar o código.

Instala o eslint globalmente:

npm install eslint -g

Inicia o repositório NPM no local:

npm init

Inicia o eslint no repositório:

eslint --init

O formato do Airbnb é bem popular e o formato de saída pode ser em json.

Após a inicialização, será gerado um arquivo .eslintrc.json, onde podem ser adicionadas regras.

Instalar extensão ESLint do VS Code.

Quando um erro ocorre, o nome da regra vem entre parênteses, exemplo:

Unexpected console statement. eslint(no-console)

Nesse caso, por exemplo, bastaria adicionar o seguinte no arquivo de configurações:

"rules": {

"no-console": 0

}

Para tornar o prettier compat´tvel com o eslint, pode instalar o seguinte pacote:

npm install eslint-config-prettier

Bastando adicionar o prettier em extends do arquivo .eslintrc.json.

**Webpack**

Consiste em um bundle, ele serve parar agrupar/processar/otimizar diversos arquivos em um único arquivo.

Para instalar, utiliza o seguinte comando:

npm install --save-dev webpack webpack-cli

(Instala o webpack e a command-line interface do mesmo)

Para utilizar o webpack em um arquivo, basta rodar o seguinte comando:

npx webpack --entry ./js/script.js -o ./

Irá agrupar todo o código, otimizar e mais. **Utilizar npx é a mesma coisa que utilizar node\_modules/.bin/webpack. Facilita para utilizarmos cli's instaladas localmente ao invés de globalmente.**

Pode também adicionar a flag watch para que ele fique atualizando o arquivo a cada modificação:

npx webpack --entry ./js/script.js -o ./ **--watch**

**NPM SCRIPTS**

Permite definirmos uma linha de comando inteira, que será ativada com **npm run nomeScript**. Não precisamos utilizar o npx aqui.

"scripts": {

"dev": "webpack --mode development --watch --entry ./js/script.js -o ./",

"build": "webpack --mode production --entry ./js/script.js -o ./"

},

(Serve para não ter que digitar esse comando todas as vezes)

**SCRIPTS EXTERNOS**

Podemos facilmente importar scripts externos instalando os mesmos através do NPM e utilizando o Webpack para fazer o bundler final.

Instala os módulos:

npm install jquery

npm install lodash

Importa os módulos:

import $ from 'jquery';

import \_ from 'lodash';

$('nav').hide();

\_.difference(['Banana', 'Morango', 'Uva'], ['Banana', 'Morango', 'Pêra']);

**BABEL**

Consiste em um compilador que transforma código novo em código antigo. Exemplo:

const nome = 'Andre'; 🡺 var nome = 'Andre';.

**POLYFILL VS TRANSPILER**

Polyfill

Cria métodos / funções com o mesmo nome das atuais, porém utilizando código antigo para permitir o uso em browsers que não possuem a API (cria uma função fetch, por exemplo, implementada com JS antigo).

Transpiler

Transforma código novo em código antigo. Ou seja, transforma const em var, por exemplo.

**INSTALAR O BABEL**

https://babeljs.io/docs/en/usage

Instala o Babel, as configurações padrões e o loader para o Webpack:

npm install --save-dev @babel/core @babel/preset-env @babel/plugin-transform-runtime babel-loader

**WEBPACK.CONFIG.JS**

Precisamos configurar o webpack, para utilizarmos o babel:

*const* path = require('path');

module.exports = {

entry: './js/script.js',

output: {

path: path.resolve(\_\_dirname, './'),

filename: 'main.js',

},

module: {

rules: [

{

test: /\.js$/,

exclude: /node\_modules/,

use: {

loader: 'babel-loader',

options: {

presets: ['@babel/preset-env'],

plugins: ['@babel/plugin-transform-runtime'],

},

},

},

],

},

};

Como os arquivos de entrada e de saída já foram definidos, eles podem ser removidos dos comandos dos scripts:

"scripts": {

"dev": "webpack --mode development --watch",

"build": "webpack --mode production"

},

**GIT**

Configurar Nome:

$ git config --global user.name "Seu Nome"

Configurar Email:

$ git config --global user.email "email@gmail.com"

Inicia um repositório:

$ git init

Adiciona o arquivo style.css ao index do git:

$ git add style.css

Com o **$ git add -A**, adicionamos todos os arquivos, inclusive apagando os arquivos removidos.

Já o **$ git add -A**, adicionamos todos os arquivos, mas não remove os arquivos.

Mostra os arquivos que tiveram mudanças.

$ git status

Irá fazer fazer o commit do código adicionado com uma mensagem.

$ git commit -m 'Descrição'

Adicionar diretório remoto

$ git remote add origin https://github.com/seuusuario/seurepositorio.git

Push do primeiro commit

$ git push -u origin master

Para listar os branchs existentes:

$ git branch

Para criar um novo branch:

$ git branch nomebranch

Para mudar o branch atual:

$ git checkout nomebranch

Sempre crie um branch

Toda funcionalidade nova, crie um branch para desenvolver a mesma. $ git checkout -b feature1

Após o desenvolvimento e commit, vá até o master e veja se existem mudanças

$ git checkout master e $ git pull

Volte para o branch e dê um merge com o master

$ git checkout feature1 e depois $ git merge master

Conflitos

Se existirem conflitos você será avisado e deverá lidar com os mesmos

Git Push

Após lidar com os conflitos faça o push do branch: $ git push e $ git push --set-upstream origin feature1.

**EVENTOS DENTRO DE CLASSES**

Quando tiver um método de uma classe que será atribuído a um event listener, o this dessa função fará referência ao elemento que o evento foi adicionado, uma opção é envolver a função em uma arrow function para que o this seja a classe.

Uma outra opção é mudar o this da função para o this da classe, o que pode ser feito adicionando o seguinte no construtor da classe:

**this. scrollToSection=this.scrollToSection.bind(this);**

Para executar um método estático dentro da própria classe, basta chamara da seguinte forma: this.constructor.incrementarNumero(); ou NomeDaClasse.incrementarNumero();

**Debounce**

O debounce consiste em uma função que limita a quantidade de execuções de uma outra função ao longo do tempo, isso melhora o desempenho. Ele elimina as funções anteriores até que sobre apenas uma que será então executada.

function debounce(callback, delay) {

    let timer;

    return (...args) => {

        if(timer) clearTimeout(timer);

        timer = setTimeout(() => {

            callback(...args);

            timer = null;

        }, delay);

    }

}

function onScroll() {

    console.log('teste');

}

const debauncedScroll = debounce(onScroll, 200);

window.addEventListener('scroll', debauncedScroll);

**Eventos personalizados**

É possível criar eventos da seguinte forma:

changeEvent = new Event( 'changeEvent' );

Para invocar o evento:

elementoQualquer.dispatchEvent(changeEvent);

**Testes**

Jasmine, Jest, Mocha

**Automação**

Webpack, Gulp, Parcel

**Frameworks**

Vue, React, Angular