JavaScript Essêncial

▼ [Arrays] - Atribuição via Desestruturação

```
// Rest (Resto) Operator
const  numeros = [1000 , 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000];
const [primeiro, segundo, ...resto] = numeros; //REST Operator
//const [primeiro, ,terceiro, ...resto] = numeros; //Posso pular elementos
console.log(primeiro, segundo );
console.log(resto);

// Spread (Espalhar) Operator - É OUTRA COISA.
```

▼ {Objetos} - Atribuição via Desestruturação

```
const pessoa = {
  //nome: 'Leomar',
  sobrenome: 'Sartor',
  idade: 30,
  endereco:{
    rua: 'Av. Toronto',
    numero: 99
  }
}

const {
  nome = 'Não existe', //Valor padrão se não existir
  sobrenome,
  idade: age, //mudar o nome da variavel para age
  endereco: { rua, numero }
} = pessoa;

console.log(nome, sobrenome, age); //Saída: Não existe Sartor 30
console.log(age); //Saída: 30
console.log(rua, numero); //Saída: Av. Toronto 99
```

▼ Funções

```
// Delcaração de função Normal - Function hoisting
falaOla(); //Saída: Olá

function falaOla(){
  console.log('Olá');
}

// Objetos de primeira Classe - Firt-Class Objects
// Function Expression
const souUmDado = function (){
  console.log('sou um dado.');
```

```
}
souUmDado(); //Saída: sou um dado.

function executaFuncao(funcao){
  funcao();
};

executaFuncao(souUmDado); //Saída: sou um dado.

// Arrow Function
const funcaoArrow = () => {
  console.log('Sou uma arrow function');
};

funcaoArrow(); //Saída: Sou uma arrow function.

//Dentro de um objeto
const obj = {
  falar() {
    console.log('Estou falando...');
  }
};
obj.falar(); //Saída: Estou falando...
```

▼ Array Filter

```
//Retornar os números maiores que dez - traz um novo array, não altera o original
const numeros = [5, 50, 80, 1, 2, 3, 5, 8, 7, 11, 15, 22, 27];

//valores implicitos - não preciso usar todos
function callbackFilter(valor, indice, array){
    return  valor > 10;
    //return true; // Embutir no novo array
    //return false; // Não embutir
}
const numerosFiltrados = numeros.filter(callbackFilter);

//Função anonima
const numerosFiltrados = numeros.filter(function(valor){
  return valor > 10;
});

//Arrow Function Abreviada
const numerosFiltrados = numeros.filter( valor => valor > 10);

console.log(numerosFiltrados );
//Saída: [50, 80, 11, 15, 22, 27]
```

```
//Retornar as pessoas que tem o nome com 5 letras ou mais
const pessoas = [
  { nome: 'Luiz', idade: 62 },
  { nome: 'Maria', idade: 23 },
  { nome: 'Eduardo', idade: 55 },
  { nome: 'Leticia', idade: 19 },
```

```
  { nome: 'Rosana', idade: 32 },
  { nome: 'Wallace', idade: 47 },
];

const pessoasComNomeGrande = pessoas.filter(pessoa => pessoa.nome.length >= 5);
//Saída: [
//  { nome: 'Eduardo', idade: 55 },
//  { nome: 'Leticia', idade: 19 },
//  { nome: 'Rosana', idade: 32 },
//  { nome: 'Wallace', idade: 47 }
//]
```

▼ Array Map

```
//Retornar os números maiores que dez - traz um novo array, não altera o original
const numeros = [5, 50, 80, 1, 2, 3, 5, 8, 7, 11, 15, 22, 27];

const numerosDobrados = numeros.map( (num) =>  num * 2 );

console.log(numerosDobrados);
//Saída: [10, 100, 160, 2, 4, 6, 10, 16, 14, 22, 30, 44, 54]
```

```
//Retornar só o nome da pessoa
const pessoas = [
  { nome: 'Luiz', idade: 62 },
  { nome: 'Maria', idade: 23 },
  { nome: 'Eduardo', idade: 55 },
  { nome: 'Leticia', idade: 19 },
  { nome: 'Rosana', idade: 32 },
  { nome: 'Wallace', idade: 47 },
];

const soNomePessoas = pessoas.map(pessoa => pessoa.nome);
//Saída: [ 'Luiz', 'Maria', 'Eduardo', 'Leticia', 'Rosana', 'Wallace' ]
const soIdades = pessoas.map(pessoa => ({ age: pessoa.idade }));
//Saída: [  { age: 62 },  { age: 23 },  { age: 55 },  { age: 19 },  { age: 32 },  { age: 47 }]

const comId = pessoas.map((pessoa, index) => ({
    id: index,
    age: pessoa.idade
}));
//Saída:
// [
//     { nome: 'Luiz', idade: 62 },
//     { nome: 'Maria', idade: 23 },
//     { nome: 'Eduardo', idade: 55 },
//     { nome: 'Leticia', idade: 19 },
//     { nome: 'Rosana', idade: 32 },
//     { nome: 'Wallace', idade: 47 }
//]
```

▼ Array Reduce (Reduzir a um elemento)

```
const numeros = [5, 50, 80, 1, 2, 3, 5, 8, 7, 11, 15, 22, 27];

const total = numeros.reduce(function (acumulador, valor, indice, array) {
    return acumulador += valor;
}, 0);
// Valor inicial pra inicial o acumulador,
// se não setar, assume o valor do primeiro elemento (valor 5)

// Saída: 236
```

```
//Retornar a pessoa mais velha
const pessoas = [
  { nome: 'Luiz', idade: 62 },
  { nome: 'Maria', idade: 23 },
  { nome: 'Eduardo', idade: 55 },
  { nome: 'Leticia', idade: 19 },
  { nome: 'Rosana', idade: 32 },
  { nome: 'Wallace', idade: 47 },
];

const maisVelha = pessoas.reduce((acc, val) => {
    if (acc.idade > val.idade)
        return acc;
    return val;
});

// { nome: 'Luiz', idade: 62 }
```

▼ Filter + Map + Reduce

```
const numeros = [5, 50, 80, 1, 2, 3, 5, 8, 7, 11, 15, 22, 27];
const numerosAlterados = numeros
  .filter(vl => vl % 2 === 0)
  .map(vl => vl *2)
  .reduce((sc, vl) = ac + valor);

// Saída: 324
```

▼ Foreach (Só para Arrays)

```
const array = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100];

array.forEach(function(valor, indice, array){
  console.log(valor, indice, array);
});
```

▼ Classes

```
class Pessoa {
  constructor(nome, sobrenome){
    this.nome = nome;
    this.sobrenome = sobrenome;
  }

  falar(){
    console.log(`${this.nome} está falando`);
  }

  comer(){
    console.log(`${this.nome} está comendo`);
  }

  beber(){
    console.log(`${this.nome} está bebendo`);
  }
}

const pessoa = new Pessoa('Leomar', 'Sartor');
```

▼ Métodos Estáticos - não tem acesso aos dados da instância

```
class Televisao{
  constructor(tv){
    this.tv = tv;
    this.volume = 0;
  }

  //Método de instância
  aumentarVolume(){
    this.volume += 1;
  }

  diminuirVolume(){
    this.volume -= 1;
  }

  //Método Estático
  static Somar( valorUm, valorDois ){
    return valorUm + ValorDois;
  }
}

const tv= new Televisao('LG');
tv.aumentarVolume();

console.log(tv);

console.log(Televisao.Somar(2,4));
```

▼ Promises

```
// 1 - O problema (Uma função não espera a outra)
//Tem 3 estados: Pending | FullFilled | Rejected
function rand(min, max) {
    min *= 1000;
    max *= 1000;
    return Math.floor(Math.random() * (max - min) + min);
}


//Descomentar pra ver o resultado
// function esperaAi(msg, temp) {
//     setTimeout(() => {
//         console.log(msg);
//     }, temp);
// }

//Descomentar pra ver o resultado
// esperaAi('Frase 1', rand(1, 3));
// esperaAi('Frase 2', rand(1, 3));
// esperaAi('Frase 3', rand(1, 3));

// 2 - Solução com CallBack (Força uma função esperar a outra)
// É um HADUKEN (Árvore de Natal) - Gambiarra

function rand(min, max) {
    min *= 1000;
    max *= 1000;
    return Math.floor(Math.random() * (max - min) + min);
}


//Descomentar pra ver o resultado
// function esperaAi(msg, temp, callbackfunction) {
//     setTimeout(() => {
//         console.log(msg);

//         if (callbackfunction) callbackfunction();
//     }, temp);
// }

//Descomentar pra ver o resultado
// esperaAi('Frase 1', rand(1, 3), function () {
//     esperaAi('Frase 2', rand(1, 3), function () {
//         esperaAi('Frase 3', rand(1, 3));
//     })
// });

// 3 - Solução com Promisses - Não precisa do HADUKEN
function esperaAi(msg, tempo) {

    // RESOLVE executa THEN()
    return new Promise((resolve, rejeita) => {
        // REJECT executa CATCH()
        if (typeof msg !== 'string') rejeita('VALOR DE ENTRADA INCORRETO');

        setTimeout(() => {
            resolve(msg);
```

```
        }, tempo);
    });
}

esperaAi('Frase 1', rand(1, 3))
    .then(resposta => {
        console.log(resposta);
        return esperaAi('Frase 2', rand(1, 3));
    })
    .then(response => {
        console.log(response);
        //Forçar o erro
        return esperaAi(429, rand(1, 3));
        //return esperaAi('Frase 3', rand(1, 3));
    })
    .then(res => {
        console.log(res);
        return res + ' LEOMAR';
    })
    .then(r => {
        console.log(r);
        console.log('FIM');
    })
    .catch(e => {
        console.log('ERROR:', e);
    });
```

▼ Métodos para trabalhar com Promisses

```
function rand(min, max) {
    min *= 1000;
    max *= 1000;
    return Math.floor(Math.random() * (max - min) + min);
}

function esperaAi(msg, tempo) {

    // RESOLVE executa THEN()
    return new Promise((resolve, rejeita) => {
        // REJECT executa CATCH()
        if (typeof msg !== 'string') {
            rejeita('CAIU NO ERRO!');
            return;
        }
        console.log(msg);
        setTimeout(() => {
            resolve(msg);

            return;
        }, tempo);
    });
}

//ALL - Respeita a Ordem - Resolve todas - se cair no erro rejeita todas
// const promises = [
//     'Primeiro valor',
```

```
//     esperaAi('Promise 1', 3000),
//     esperaAi('Promise 2', 500),
//     esperaAi('Promise 3', 1000),
//     //esperaAi(1000, 1000),
//     'Outro valor',
// ];

// Promise.all(promises)
//     .then(function (vlr) {
//         console.log(vlr);
//     })
//     .catch(function (err) {
//         console.log(err);
//     });

//RACE - Corrida - a primeira que resolver me entrega o valor de promessa
//isso não empede dele resolver as outras promessa. Apenas entrega só a primeira
const promises2 = [
    //'Primeiro valor', // Não é uma promess, mas entende com promessa resolvida
    esperaAi('Promise 1', 3000),
    esperaAi('Promise 2', 500),
    esperaAi('Promise 3', 1000),
    //'Outro valor',
];

Promise.race(promises2)
    .then(function (vlr) {
        console.log(vlr);
    })
    .catch(function (err) {
        console.log(err);
    });

//RESOLVE - Para verificar dados por exemplo CACHE se já tiver entrega se não faz o processamento
//REJECT - Força o reject na hora
```

▼ Async / Await (melhora a legibilidade das promisses)

```
//Permitem utilização de promisses como se fosse síncronas, ou seja
//aguardar uma promisse terminar
async function executa(){
  try{
    const valor1 = await esperaAi('Fase 1', rand(1, 3));
    console.log(valor1);
    const valor2 = await esperaAi('Fase 2', rand(1, 3));
    console.log(valor2);
    const valor3 = await esperaAi('Fase 3', rand(1, 3));
    console.log(valor3);
  }catch(e){
    console.log(e)
  }
}

executa();
```

▼ **Fetch API (Retorna Promisses - Melhoria pra XML Http Request)**

▼ Fetch API e Axios (JSON)