Aprenda JavaScript moderno (ES6+) para front-end e back-end

Este documento inclui aprendizado sobre JavaScript e TypeScript, você vai aprender ambas as linguagens no mesmo pacote.

Inicialmente, você aprenderá a utilizar recursos nativos do JavaScript sem a necessidade de utilizar frameworks ou bibliotecas adicionais.Trabalharemos tanto no Node.Js (back-end) quanto no navegador (front-end).

1. JavaScript Funçoes

Declaração de funções

- Function hoisting

O motor do Javascript eleva as declarações de funções e variáveis para o topo do JavaScript, isso significa que você pode fazer a chamada da função tanto antes ou depois da declaração da função.

Ex: chamada pós declaração.

```
// Declaração de função (Function hoisting)
function falaOi(){
    console.log('Oie');
};
falaOi();
```

Ex: Chamada antes da declaração

```
// Declaração de função (Function hoisting)
falaOi();
function falaOi(){
    console.log('Oie');
};
```

❖ Function expression

A função pode ser tratada como dado neste tipo de função ao criar uma função a mesma pode ser tratada como dado sentando em uma variável.

Ex: Tratando função como dado e passando a função como parâmetro.

```
// First-class objects (Objetos de primeira classe)

// Function expression
const souUmDado = function(){
    console.log('Sou um dado.');
};
souUmDado();
function executaFuncao(função){
    console.log('Vou executar sua função abaixo:');
    função();
}
executaFuncao(souUmDado);
```

❖ Arrow function

Trata-se um recurso novo do **ES6 é** uma declaração de função mais curta sem a necessidade da palavra function, porém, o resultado é o mesmo.

Ex: Arrow function

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

❖ Function dentro de um objeto

A função dentro de um objeto é tratada como se ela fosse um método do objeto.

Ex: Função dentro do objeto

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

Parâmetros da função

Toda função que contém a palavra reservada function, ou seja, toda função que não seja uma função Arrow tem por default um parâmetro chamado arguments com isso mesmo que sua função não espere por nenhum parâmetro em sua declaração ela aceita envio de argumentos na sua chamada. Agora caso uma função esteja esperando um parâmetro e não for enviado nenhum argumento em sua chamada à função por default irá definir o valor como undefined e não retornará erro. Se uma função esperar por um parâmetro que será utilizado para fazer um calculo você pode setar um valor default caso não for enviado isso faz com que a função não retorne NaN e faça o calcula a partir do valor default também podemos utilizar o rest parameter, porém este parâmetro deve ser sempre o ultimo na ordem de utilização.

Ex: Parâmetro arguments

```
// Toda função tem um argumento chamado arguments que sustenta todos argumentos
enviados
function funcao(a, b, c){
    console.log(arguments)
}
funcao(1,2,3,4,5,6,7);

function funcao1(a, b, c){
    let total = 0;
    for (let argumento of arguments) {
        total += argumento;
    }
    console.log(total,a,b,c);
}
 funcao1(1,2,3,4,5,6,7);
```

Ex: Função declarada com parâmetro e não foi enviado argumento

```
//Quando a função tem parametro e não são envidos os argumentos assume o valor de
undefined
function funcao2(a,b,c,d,e,f){
    console.log(a,b,c,d,e,f);
}
funcao2(1, 2, 3)
```

Ex: Parametro default

```
//Parametro com valor default
function funcao3(a = 1, b = 2, c = 5){
    console.log(a+ b + c);
}
funcao3(2,3)
```

Ex: Rest parameter

```
//Rest parameter
function conta(operador,acumulador,...numeros){
    for(let numero of numeros){
        if(operador == '+') acumulador += numero;
        if(operador == '-')acumulador -= numero;
        if(operador == '/')acumulador /= numero;
        if(operador == '*')acumulador *= numero;
    }
    console.log(acumulador);
}
conta('*',1,20,30,40,50)
```

Função callback

Funções de callback servem para manter a ordem de execução das funções digamos que para o funcionamento correto do seu processo uma função tenha que esperar a o término de uma outra com as funções de call-back isso é viável supondo que o servidor responda as requisições em tempos diferentes, porém você tem uma ordem para execução das funções com as funções de callback você garante a ordem de execução. Existe outra forma de fazer garantir a ordem de execução com promises que será explicado mais a frente.

Ex:

```
function rand(min = 1000, max = 3000){
    const num = Math.random() * (max - min) + min;
    return Math.floor(num);
}

function f1(callback) {
    setTimeout(function() {
        console.log('f1');
        if(callback) callback();
    },rand());
}
```

```
function f2(callback) {
    setTimeout(function(){
        console.log('f2');
        if(callback) callback();
    },rand());
}

function f3(callback) {
    setTimeout(function(){
        console.log('f3');
        if(callback) callback();
    },rand());
}

f1(f1Callback);

function f1Callback(){
    f2(f2Callback);
}

function f2Callback(){
    f3(f3Callback);
}

function f3Callback(){
    console.log('Cheguei');
}
```

Funções imediatas(IIFE – Immediately Invoke Function Expression)

Funções imediatas são funções que ecapsulam as vaiáveis ou funções da função diferenciando das variáveis do scopo global isso permite você ter variáveis com o mesmo nome dentro do scopo da sua função sem entrarem em conflito com as variáveis do scopo global. As funções imediatas é uma função encapsulada por parênteses e é chamada por abertura e fechamento de parênteses logo após se termino podendo conter parâmetros ou não.

Ex:

```
// Immediately invoke function expression
(function(idade, peso, altura){
    const sobreNome = 'Abreu';
    function criaNome(nome){
        return nome + ' ' + sobreNome;
    }
```

```
    function falaNome(){
        console.log(criaNome('Jorge'));
    }

    falaNome();
    console.log(idade, peso, altura);
})(42, 90, 1.82);
```

Função fábrica(Factory Functions)

Função fábrica é responsável por definir um objeto isso é bem parecido com o conceito de classe, porém não trata-se de uma classe veremos classes mais à frente. Na Função fábrica você pode criar métodos e variáveis normalmente.

Ex:

```
function criaPessoa(nome, sobreNome, altura, peso){
    return {
        nome,
        sobreNome,
        altura,
        peso,

        // Getter
        get nomeCompleto(){
            return '${this.nome} ${this.sobreNome}';
        },

        // Setter
        set nomeCompleto(valor){
            valor = valor.split(' ');
            this.nome = valor.shift();
            this.sobreNome = valor.join(' ');
        },

        fala(assunto){
            return '${this.nome} está ${assunto}.';
        },
        get imc(){
            const indice = this.peso / (this.altura * this.altura);
            return indice.toFixed(2);
        }
    };
}
```

```
const p1 = criaPessoa('Jorge', 'Abreu', 1.82, 82.5);
const p2 = criaPessoa('Luana', 'Silva', 1.75, 120);
const p3 = criaPessoa('Lara', 'Abreu', 1.10, 25);

console.log(p1.imc);
console.log(p2.imc);
console.log(p3.imc);
```

Função construtora

Conforme a função fábrica a função construtora é a matriz de um objeto, porém na função construtora o conceito é muito mais parecido com o conceito de classe por convenção uma função construtora inicia-se com letra maiúscula e também para criar um objeto é necessário iniciar com a palavra new como instancia uma classe, outra diferença é que nas funções construtoras não contém returno.

Ex:

```
// Função construtora -> objetos

// Função fabrica -> objetos
// Construtora -> Pessoa (new)
function Pessoa(nome, sobrenome){
    // Atributos ou métodos privados
    const Id = 12345;
    const metodoInterno = function(){

    };

    // Atributos ou métodos publicos
    this.nome = nome;
    this.sobrenome = sobrenome;

    this.metodo = function(){
        console.log(this.nome + ': sou um método');
    }
}

const p1 = new Pessoa('Jorge', 'Abreu');
const p2 = new Pessoa('Luana', 'Rocha');

console.log(p1.nome);
p1.metodo();
```