Seção 4: JavaScript Funções (Avançado)

* Aula 49 maneiras de declarar funções

// Declaração de função (Function Hoisting)

falaOi();    //pode ser chamado tanto antes como depois

function falaOi(){

    console.log('Oi');

}

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

// Function expression

const souUmDado = function() {

    console.log('Sou um dado');

};

function executaFuncao(funcao) {

    funcao();

}

executaFuncao(souUmDado)

// Arrow function

const funcaoArrow = () => {

    console.log('Sou uma arrow function');

};

funcaoArrow();

// Dentro de um objeto

const obj = {

    falar(){

        console.log('Estou falando...');

    }

}

obj.falar();

uma variável pode se tornar uma função.

* Aula 50 Parâmetro de Funções

function funcao(){

    console.log(arguments);

}

funcao('Valor', 1, 2, 3, 4, 5);

[Arguments] {

 '0': 'Valor', '1': 1, '2': 2, '3': 3, '4': 4, '5': 5 }

quando chamado uma função com function, os argumentos são passados para o atributo arguments.(com arrow function não funciona)

function funcao(){

    let total = 0;

    for(let argumento of arguments){

        total += argumento

    }

    console.log(total);

}

funcao(1, 2, 3, 4, 5, 6, 7);

esses argumento podem ser acessados

function funcao(a, b, c, d, e, f){

    console.log(a, b, c, d, e, f);

}

funcao(1, 2, 3);

quando passado menos argumentos do que paramentro e esses parâmetro forem iniciado, os que não tiverem valor serão undefined

function funcao(a, b, c, d, e, f){   // menos argumentos

 que parametro retorna undefied

    console.log(a, b, c, d, e, f);   // naqueles que não foram passado

}

funcao(1, 2, 3);

menos argumento que parâmetros, so retorna os valores passado

function funcao(a, b = 2, c = 4){ // b como padrão será

  2, caso não haja algum

    console.log(a + b + c);

}

funcao(2, undefined, 20);  // para deixar o valor padrão de b

para que o valor assuma o seu valor padrão e enviado o undefined

function funcao({ nome, sobrenome, idade }){

    console.log(nome, sobrenome, idade);

}

funcao( { nome:'Luiz', sobrenome:'Otávio', idade: 30 });

função por desestruturação

function funcao({ nome, sobrenome, idade }){

    console.log(nome, sobrenome, idade);

}

let obj = { nome: 'Luiz', sobrenome: 'Otavio', idade: 30 };

console.log(obj);

ou pode ser enviado direto com o objeto

function funcao([valor1, valor2, valor3]) {

    console.log(valor1, valor2, valor3);

}

funcao(['Luiz Otávio', 'Miranda', 30])

passando por Array

function conta(operador, acumulador, ...numeros){  // rest opera

tor - sempre deve ser o ultimo

    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);

const conta = (...args) => {   // para pegar os argumentos com

arrow function

    console.log(args);

}

conta('+', 1, 20, 30, 40, 50);

para conseguir pegar os argumentos com arrow function

* Aula 51 Retorno da função

// return

// Retorna um valor

// Termina a função

function soma(a , b){

    return a + b; // retorna algo

}

let s1 = soma(2, 5);

function soma2(a, b){

    console.log(a + b); // Não retorna nada

}

soma2(5, 2);

function criaPessoa(nome, sobrenome){

    return { nome: nome, sobrenome: sobrenome };

}

const p1 = criaPessoa('Luiz', 'Otavio');

const p2 = {

    nome: 'João',

    sobrenome: 'Oliveira'

};

console.log(p1);

console.log(p2);

mesmo retorno

function falaFrase(comeco) {

    function falaResto( resto){

        return comeco + ' ' +resto;

    }

    return falaResto;

}

const olaMundo = falaFrase('Olá');

console.log(olaMundo('Mundo!'));

retorno encadeado um dentro do outro.

function duplica(n){

    return n \* 2;

}

function triplica(n){

    return n \* 3;

}

function quadriplica(n){

    return n \* 4;

}

console.log(duplica(2));

console.log(triplica(2));;

console.log(quadriplica(2));

veja esse caso de código que poderia ser encadeados.

function criaMultiplicador(multiplicador) {

    return function(n) {

        return n \* multiplicador;

    }

}

const duplica = criaMultiplicador(2);

const triplica = criaMultiplicador(3);

const quadriplica = criaMultiplicador(4);

console.log(duplica(3));

console.log(triplica(2));

console.log(quadriplica(10));

ficaria dessa maneira

* Aula 52 escopo léxico

const  nome = 'Luiz';

function falaNome() {

    console.log(nome);

}

function usaFalaNome(){

    const nome = 'Otavio';

    falaNome();

}

usaFalaNome();

* Aula 53 closures

// Habilidade que a função tem de acessar seu escopo léxico

function retornaFuncao(nome) {

    return function () {

        return nome;

    }

}

const funcao = retornaFuncao('Luiz');

const funcao2 = retornaFuncao('João');

console.dir(funcao);

console.dir(funcao2);

console.log(funcao(), funcao2());

* Aula 54 função calback

function rand( min = 1000, max = 3000) {

    const num = Math.random() \* (max - min) + min;

    return Math.floor(num);

}

function f1() {

    setTimeout(function() {

        console.log('f1');

    }, rand())

}

function f2() {

    setTimeout(function (){

        console.log('f2');

    }, rand())

}

function f3() {

    setTimeout(function(){

        console.log('f3');

    }, rand())

}

f1();

f2();

f3();

console.log('Olá mundo');

funções que geram um tempo aleatório para serem exibidas, porem o ola mundo sempre será o primeiro a ser executado, pois não depende de nenhum tempo.

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(function() {

    f2(function() {

        f3(function(){

            console.log('Olá mundo');

        });

    });

});

Temos aqui uma função de callback, que so sera exibida quando a anterios for acabar de ser executada.

Ou pode ser assim

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('Olá mundo');

}

* Aula 55 funções imediatas (IIFE) ou funções autoinvocadas

//IIFE -> Immediately invoked function expression

function qualquerCoisa(){

    console.log(123456);

}

qualquerCoisa()

aqui essa função toca o escopo global quando ela é chamada.

//IIFE -> Immediately invoked function expression

(function() {

    console.log(123456);

})()

Uma função que não toca o escopo global, ficando protegido. Pois fica sendo um função anônima.

//IIFE -> Immediately invoked function expression

(function(idade, peso, altura) {

    const sobrenome = 'Miranda';

    function criaNome(nome){

        return nome + ' ' + sobrenome;

    }

    function falaNome(){

        console.log(criaNome('Luiz'));

    }

    falaNome();

    console.log(idade, peso, altura);

})(30, 80, 1.80);

Como ficaraia essa função protegida que não toca o escopo global

* Aula 56 funções fabrica (factory function)

function criaPessoa(nome, sobrenome){

    return {

        nome, sobrenome

    };

}

const p1 = criaPessoa('Rodrigo', 'Souza');

console.log(p1);

criando um objeto

function criaPessoa(nome, sobrenome){

    return {

        nome,

        sobrenome,

        fala: function(assunto){

            return `${nome} está ${assunto}`

        }

    };

}

const p1 = criaPessoa('Rodrigo', 'Souza');

console.log(p1.fala('falando sobre JS'));

modelo de objetos com funções, chamasse de método

function criaPessoa(nome, sobrenome){

    return {

        nome,

        sobrenome,

        fala: function(assunto){

            return `${peso} está ${assunto}`

        },

        peso: 80

    };

}

const p1 = criaPessoa('Rodrigo', 'Souza');

console.log(p1.fala('falando sobre JS'));

nesso modelo acima ocorreria um erro , porque a função não acharia o atributo peso, para resolver esse problema uma solução séria colocar a palavra this.peso, que iria pegar o peso de que chamou o this

function criaPessoa(nome, sobrenome){

    return {

        nome,

        sobrenome,

        fala(assunto){

            return `${ this. nome} está ${assunto}`

        },

    };

}

const p1 = criaPessoa('Rodrigo', 'Souza');

console.log(p1.fala('falando sobre JS'));

console.log(p1.nome);

nesse caso o this e quem chamou ele, this está dentro da função fala(), e quem chamou a função fala(), foi o p1, então this e o p1.

Seria a mesma coisa que fala p1.nome

// Factory function (Função fábrica)

// Constructor function (Função construtora)

function criaPessoa(nome, sobrenome, altura, peso){

    return {

        nome,

        sobrenome,

        fala(assunto){

            return `${this.nome} está ${assunto}`

        },

        altura: altura,

        peso: peso,

        imc() {

            const indice = this.peso / (this.altura \*\* 2);

            return indice.toFixed(2);

        }

    };

}

const p1 = criaPessoa('Rodrigo', 'Souza', 1.80, 80);

console.log(p1.imc());

const p2 = criaPessoa('Maria', 'Joaquina', 1.60, 42);

console.log(p2.imc());

aqui temos um método que pega o peso e altura e faz o imc de cada objeto com a ajuda do this.

function criaPessoa(nome, sobrenome, altura, peso){

    return {

        nome,

        sobrenome,

        fala(assunto){

            return `${this.nome} está ${assunto}`

        },

        altura: altura,

        peso: peso,

        get imc() {

            const indice = this.peso / (this.altura \*\* 2);

            return indice.toFixed(2);

        }

    };

}

const p1 = criaPessoa('Rodrigo', 'Souza', 1.80, 80);

console.log(p1.nome);

console.log(p1.sobrenome);

console.log(p1.imc);

colocando o get na frente do método desse objeto, esse método agora passa a fingir como se fosse um atributo, não prescisando usar as (), para quando for chamado.

Passando a ser chamado como um atributo normal, porem ainda e um método, na programação isso e chamado de GETTER.

 //Setter

 set nomeCompleto(valor) {

   valor = valor.split(' '); //dividindo a string por espaços

   this.nome = valor.shift(); // pegando o primeiro indice

   this.sobrenome = valor.join(' '); //juntando os indice com

espaço

},

p1.nomeCompleto = 'Rodrigo de Souza Silva';

aqui estamos ultilizando o set, para setar valor, aqui o valor e passado pra uma variável, que divide ela em uma Array cada vez que encontra um espaço, depois pega o primeiro valor desse Array e põem em this.nome, em sequida junta o restante da Array separando cada Array por espaço e joga esse valor em this.sobrenome.

function criaPessoa(nome, sobrenome, altura, peso){

    return {

        nome,

        sobrenome,

        // Getter

        get nomeCompleto() {

            return `${this.nome} ${this.sobrenome}`;

        },

        //Setter

        set nomeCompleto(valor) {

            valor = valor.split(' '); //dividindo a string por espaços

            this.nome = valor.shift(); // pegando o primeiro indice

            this.sobrenome = valor.join(' '); //juntando os indice com espaço

        },

        fala(assunto){

            return `${this.nome} está ${assunto}`

        },

        altura: altura,

        peso: peso,

        // Getter

        get imc() {

            const indice = this.peso / (this.altura \*\* 2);

            return indice.toFixed(2);

        }

    };

}

const p1 = criaPessoa('Luiz', 'Otavio', 1.80, 80);

const p2 = criaPessoa('Maria', 'Otavio', 1.30, 90);

const p3 = criaPessoa('João', 'Otavio', 1.60, 110);

console.log(p1.nome, p1.sobrenome);

p1.nomeCompleto = 'Rodrigo de Souza Silva';

console.log(p1.nome, p1.sobrenome);

console.log(p1.imc);

console.log(p2.imc);

console.log(p3.imc);

código completo para estudos.

* Aula 58 funçoes construtoras (constructor functions)

// Função construtora -> constroi obetos

// Função fabrica -> fabrica objetos

// Construtora -> Pessoa (new) = new Pessoa = o nome prescisa ser em maiusculos

function Pessoa(nome, sobrenome) {

    this.nome = nome,

    this.sobrenome = sobrenome

}

Como criar uma função construtora

function Pessoa(nome, sobrenome) {

    this.nome = nome,

    this.sobrenome = sobrenome

}

const p1 = new Pessoa('Luiz', 'Otavio');

const p2 = new Pessoa('Maria', 'Oliveira');

criamos agora duas pessoas, p1 e p2, que receberam seus nome e sobrenome

function Pessoa(nome, sobrenome) {

    this.nome = nome,

    this.sobrenome = sobrenome,

    this.metodo = () => {

        console.log('Sou um método');

    }

}

const p1 = new Pessoa('Luiz', 'Otavio');

const p2 = new Pessoa('Maria', 'Oliveira');

p1.metodo();

criando e chamando um método.

function Pessoa(nome, sobrenome) {

    // Atributos ou Métodos Privados

    const ID = 123456;

    const metodoInterno = function(){

        console.log('Não poderá ser chamada');

    };

    // Atributos ou Métodos publicos

    this.nome = nome;

    this.sobrenome = sobrenome;

    this.metodo = () => {

        console.log('Sou um método');

    };

}

const p1 = new Pessoa('Luiz', 'Otavio');

const p2 = new Pessoa('Maria', 'Oliveira');

console.log(p1.ID); // ERRO

não estará disponível para acessar através do ponto, só estará disponível somente internamente. Esses são os atributos privados.

* Aula 59 exercicio calculadora por função construtora

function Calculadora(){

  this.display = document.querySelector('.display');

  this.inicia = () => {

    this.capturaCliques();

    this.capturaEnter();

  };

  this.capturaCliques = () => {

    document.addEventListener('click', event => {

      const el = event.target;

      if(el.classList.contains('btn-num')) this.addNumDisplay(el);

      if(el.classList.contains('btn-clear')) this.clear();

      if(el.classList.contains('btn-del')) this.del();

      if(el.classList.contains('btn-eq')) this.realizaConta();

    });

  };

  this.addNumDisplay = (el) => {

    this.display.value += el.innerText;

    this.display.focus();

  };

  this.clear = () => this.display.value = '';

  this.del = () => this.display.value = this.display.value.slice(0, -1);

  this.realizaConta = () => {

    try{

      const conta = eval(this.display.value);

      if(!conta) {

        alert('Conta inválida');

        return;

      }

      this.display.value = conta;

    } catch(e) {

      alert('Conta inválida');

    }

  };

  this.capturaEnter = () => {

    document.addEventListener('keyup', e => {

      if(e.keyCode === 13) {

        this.realizaConta();

      }

    });

  }

}

const calculadora = new Calculadora();

calculadora.inicia();

* Aula 60 funções recursivas

function recursiva(max) {

    console.log(max);

    if(max >= 10) return;

    max++;

    recursiva(max)

}

recursiva(0);

são funções que se chamam de volta, única coisa que precisa se preocupa e quando ela precisa parar.

* Aula 61 funções geradoras

function\* geradora1() {

    // Código qualquer ...

    yield 'Valor 1';

    // Código qualquer ...

    yield 'Valor 2';

    // Código qualquer ...

    yield 'Valor 3';

}

const g1 = geradora1();

console.log(g1.next());

na primeira fez que chama a função ela retornará um valor, na segunda vez que ela for chamada ela retornará o segundo valor e assim por diante, com o uso de .next

{ value: 'Valor 1', done: false }

Ira retorna essa Array de informação, o seu valor e o done, que verifica se já chegou no final.

function\* geradora1() {

    // Código qualquer ...

    yield 'Valor 1';

    // Código qualquer ...

    yield 'Valor 2';

    // Código qualquer ...

    yield 'Valor 3';

}

const g1 = geradora1();

console.log(g1.next().value); // valor 1

console.log(g1.next().value); // valor 2

console.log(g1.next().value); // valor 3 done: false

console.log(g1.next());       // undefined, done: true

ele sabe quando o valor chegou ao final

const g1 = geradora1();

for (let valor of g1){

    console.log(valor);

}

function\* geradora2() {

    let i = 0;

    while(true) {       // essa geradora é infinita

        yield i;

        i++;

    }

}

const g2 = geradora2();

console.log(g2.next()); // { value: 0, done: false }

console.log(g2.next()); // { value: 1, done: false }

console.log(g2.next()); // { value: 2, done: false }

essa geradora e infinita.

function\* geradora3() {

    yield 0;

    yield 1;

    yield 2;

}

function\* geradora4() {

    yield\* geradora3();

    yield 3;

    yield 4;

    yield 5;

}

const g4= geradora4();

for(let valor of g4) {

    console.log(valor);

}

0, 1 e 2 serão gerados pelo primeiro gerador, já os demais números 3, 4 e 5 pelo segundo gerador.

function\* geradora5() {

    yield function() {

        console.log('Vim do y1');

    };

    // ....

    yield function() {

        console.log('Vim do y2');

    }

}

const g5 = geradora5();

const func1 = g5.next().value;

const func2 = g5.next().value;

func1();

func2();