Curso Web Moderno Completo com JavaScript 2021 + Projetos

Lista de ilustrações

**Nenhuma entrada de índice de ilustrações foi encontrada.**

**Nenhuma entrada de índice de ilustrações foi encontrada.**

Lista de Tabelas

[Tabela 2.1. Materiais e suas respectivas propriedades mecânicas. 27](#_Toc60871303)

Sumário

[1 Configuração do Ambiente 8](#_Toc65049579)

[1.1 Instalação Vscode 8](#_Toc65049580)

[1.2 Instalação Node 8](#_Toc65049581)

[2 Fundamentos 8](#_Toc65049582)

[2.1 Visão Geral de Algoritmo 8](#_Toc65049583)

[2.2 Visão Geral de Estruturas de Dados 8](#_Toc65049584)

[2.3 Organização Básica de um Código JS 8](#_Toc65049585)

[2.4 Executando JavaScript 9](#_Toc65049586)

[2.5 Comentários de Código 9](#_Toc65049587)

[2.6 O básico de Var, Let e Const 9](#_Toc65049588)

[2.7 Tipagem Fraca 10](#_Toc65049589)

[2.8 Tipos em JavaScript 10](#_Toc65049590)

[2.9 Number: Alguns Cuidados 11](#_Toc65049591)

[2.10 Usando o Math 11](#_Toc65049592)

[2.11 Tipos em JavaScript: Strings 12](#_Toc65049593)

[2.12 Templates Strings 13](#_Toc65049594)

[2.13 Tipos em JavaScript: Booleanos 13](#_Toc65049595)

[2.14 Tipos em JavaScript: Array 14](#_Toc65049596)

[2.15 Tipos em JavaScript: Object 15](#_Toc65049597)

[2.16 Entendendo o Null & Undefined 16](#_Toc65049598)

[2.17 Quase Tudo é Função 18](#_Toc65049599)

[2.18 Exemplos Básicos de Funções #01 18](#_Toc65049600)

[2.19 Exemplos Básicos de Funções #02 19](#_Toc65049601)

[2.20 Declaração de Variáveis com Var #01 19](#_Toc65049602)

[2.21 Declaração de Variáveis com Var #02 20](#_Toc65049603)

[2.22 Declaração de Variáveis com let 20](#_Toc65049604)

[2.23 Usando Var em Loop 21](#_Toc65049605)

[2.24 Usando Let em Loop 21](#_Toc65049606)

[2.25 Entendendo Hoisting 22](#_Toc65049607)

[2.26 Função Vs Objeto 23](#_Toc65049608)

[2.27 Par Nome/Valor 24](#_Toc65049609)

[2.28 Notação Ponto 25](#_Toc65049610)

[2.29 Operadores: Atribuição 26](#_Toc65049611)

[2.30 Operadores: Destructing #01 27](#_Toc65049612)

[2.31 Operadores: Destructing #02 28](#_Toc65049613)

[2.32 Operadores: Destructing #03 28](#_Toc65049614)

[2.33 Operadores: Destructing #04 29](#_Toc65049615)

[2.34 Operadores: Aritméticos 29](#_Toc65049616)

[2.35 Operadores: Relacionais 30](#_Toc65049617)

[2.36 Operadores: Lógicos 31](#_Toc65049618)

[2.37 Operadores: Unário 33](#_Toc65049619)

[2.38 Operadores: Ternário 33](#_Toc65049620)

[2.39 Contexto de Execução: Browser x Node 34](#_Toc65049621)

[3 Estruturas de Controle 35](#_Toc65049622)

[3.1 Usando a Estrutura IF #01 35](#_Toc65049623)

[3.2 Usando a Estrutura IF #02 36](#_Toc65049624)

[3.3 Usando a Estrutura IF/ELSE 37](#_Toc65049625)

[3.4 Estrutura IF/ELSE IF 38](#_Toc65049626)

[3.5 Usando a Estrutura Switch 38](#_Toc65049627)

[3.6 Estrutura While 39](#_Toc65049628)

[3.7 Estrutura FOR/IN 40](#_Toc65049629)

[3.8 Usando break/continue 40](#_Toc65049630)

[4 Funções 41](#_Toc65049631)

[4.1 Cidadão de Primeira Linha 41](#_Toc65049632)

[4.2 Parâmetros e retorno são opcionais 43](#_Toc65049633)

[4.3 Parâmetros Variáveis 43](#_Toc65049634)

[4.4 Parâmetro Padrão 44](#_Toc65049635)

[4.5 ‘this’ pode variar 45](#_Toc65049636)

[4.6 ‘this’ e a função bind #01 45](#_Toc65049637)

[4.7 ‘this’ e a função bind #02 46](#_Toc65049638)

[4.8 Funções Arrow #01 47](#_Toc65049639)

[4.9 Funções Arrow #02 48](#_Toc65049640)

[4.10 Funções Arrow #03 49](#_Toc65049641)

[4.11 Funções Anônimas 50](#_Toc65049642)

[4.12 Funções Callback #01 51](#_Toc65049643)

[4.13 Funções Callback #02 51](#_Toc65049644)

[4.14 Funções Callback #03 52](#_Toc65049645)

[4.15 Funções construtoras 52](#_Toc65049646)

[4.16 Tipos de Declaração 53](#_Toc65049647)

[4.17 Contexto Léxico 54](#_Toc65049648)

[4.18 Closures 55](#_Toc65049649)

[4.19 Função Factory 56](#_Toc65049650)

[4.20 Classe vs Função Factory 57](#_Toc65049651)

[4.21 Immediately Invoked Function Expression (IIFE) 58](#_Toc65049652)

[4.22 Call & Apply 58](#_Toc65049653)

[5 Objetos & OO 59](#_Toc65049654)

[5.1 Introdução à OO 59](#_Toc65049655)

[5.2 Revisão sobre Objeto 61](#_Toc65049656)

[5.3 Estratégias para Criação de Objetos 62](#_Toc65049657)

[5.4 Objetos Constantes 64](#_Toc65049658)

[5.5 Notação Literal 65](#_Toc65049659)

[5.6 Getters e Setters 66](#_Toc65049660)

[5.7 Funções Importantes de Objetos 67](#_Toc65049661)

[5.8 Herança #01 68](#_Toc65049662)

[5.9 Herança #02 69](#_Toc65049663)

[5.10 Herança #03 71](#_Toc65049664)

[5.11 Herança #04 72](#_Toc65049665)

[5.12 Herança #05 73](#_Toc65049666)

[5.13 Herança #06 74](#_Toc65049667)

[5.14 Evitando Modificações 75](#_Toc65049668)

[5.15 JSON vs Objeto 75](#_Toc65049669)

[5.16 Classe #01 76](#_Toc65049670)

[5.17 Classe #02 77](#_Toc65049671)

[6 Array 78](#_Toc65049672)

[6.1 Visão Geral 78](#_Toc65049673)

[6.2 Array: Métodos Importantes 80](#_Toc65049674)

[6.3 Simulando Array com Objeto 81](#_Toc65049675)

[6.4 Foreach #01 81](#_Toc65049676)

[6.5 Foreach #02 82](#_Toc65049677)

[6.6 Map #01 82](#_Toc65049678)

[6.7 Map #02 83](#_Toc65049679)

[6.8 Map #03 83](#_Toc65049680)

[6.9 Filter #01 84](#_Toc65049681)

[6.10 Filter #02 85](#_Toc65049682)

[6.11 Reduce #01 86](#_Toc65049683)

[6.12 Reduce #02 87](#_Toc65049684)

[6.13 Reduce #03 88](#_Toc65049685)

[7 Node 90](#_Toc65049686)

[8 ES Next... 6,7,8 90](#_Toc65049687)

[9 Fundamentos da Web 90](#_Toc65049688)

[10 HTML 90](#_Toc65049689)

[11 CSS 90](#_Toc65049690)

[12 DOM 90](#_Toc65049691)

[13 Ajax 90](#_Toc65049692)

[14 Gulp 90](#_Toc65049693)

[15 Webpack 90](#_Toc65049694)

[16 jQuery 90](#_Toc65049695)

[17 Bootstrap 90](#_Toc65049696)

[18 React 90](#_Toc65049697)

[19 Vue 90](#_Toc65049698)

[20 Banco de Dados Relacionais 90](#_Toc65049699)

[21 Banco de Dados Não Relacionais 90](#_Toc65049700)

[22 Express 90](#_Toc65049701)

[23 Mongoose 90](#_Toc65049702)

[24 Projetos 90](#_Toc65049703)

[25 Outros Tópicos 90](#_Toc65049704)

[26 Conclusão 90](#_Toc65049705)

# Configuração do Ambiente

## Instalação Vscode

## Instalação Node

# Fundamentos

## Visão Geral de Algoritmo

Algoritmo é uma sequência de passos, uma sequência de instruções a serem seguidas para sair de um ponto e ir para outro. Os algoritmos possuem algumas características:

* É possível repetir determinados passos dentro de um algoritmo;
* É possível tomar decisões;
* As entradas determinam a saída.

## Visão Geral de Estruturas de Dados

Estrutura de Dados é diferente de dados, a estrutura visa duas cosias, organizar e administrar os dados.

A Estrutura de Dados envolve os critérios como os dados são armazenados (inteiro x flutuante), (listas x filas x arvores).

A estrutura de dados básica se baseia em tipagem (real, inteiro, string, booleano, etc..).

Em JavaScript por exemplo, temos apenas uma estrutura para representação de números, o ‘Number’, diferente de python, que temos ‘Int’ e ‘float’, por exemplo.

É possível criar uma estrutura de dados personalizada, como por exemplo uma tabela de banco de dados.

## Organização Básica de um Código JS

Um código JS é organizado em sentenças de códigos que podem ou não terminar em ‘;’. Além das sentenças, um código JS também é organizado em blocos (agrupador de sentenças), que ficam dentro de parênteses ({ }).

## Executando JavaScript

Podemos executar códigos JavaScript em IDE’s como VsCode, Sublime, Atom, etc... com o Node, ou nos navegadores acessando o console, ou também em ferramentas online.

## Comentários de Código

// Comentário de uma linha

/\*

Comentário de

múltiplas linhas

\*/

## O básico de Var, Let e Const

var a = 3 // Os nomes é muito IMPORTANTE

let b = 4 // De preferência para o let

Aqui definimos duas variáveis, ‘a’ e ‘b’ com var e let respectivamente.

Se quisermos redeclarar

var a = 30

let b = 40

Vamos conseguir apenas redeclarar a variável ‘a’, pois não podemos redeclarar variáveis com o let, sendo necessário:

var a = 30

b = 40

console.log(a, b) : 30, 40

O mais indicado é declarar uma única vez e para novas atribuições, não utilizar o var:

var a = 3 // Os nomes são muito IMPORTANTE

let b = 4

a = 30

b = 40

console.log(a, b) : 30, 40

Para declarar constantes utilizamos:

const c = 5

//c = 50 Não é possível alterar, é constante

console.log(c): 5

## Tipagem Fraca

JS tem uma tipagem fraca, ou seja, mais dinâmica, flexível, porém, existem desvantagens como a dificuldade de encontrar determinados erros.

Ser uma linguagem de tipagem fraca, quer dizer que as variáveis podem ser inteiros em determinado momento, strings em outro e funções em outro, ou seja, são flexíveis.

let qualquer = 'Legal'

console.log(qualquer): Legal

console.log(typeof qualquer): string

qualquer = 3.141516

console.log(qualquer): 3.141516

console.log(typeof qualquer): number

## Tipos em JavaScript

const peso1 = 1.0 // Uma maneira de declarar números

const peso2 = Number('2.0') // Outra maneira

console.log(peso1, peso2)

console.log(Number.isInteger(peso1)) /\* Verificação se uma variável ou constante é um número inteiro

\*/

console.log(Number.isInteger(peso2))

const avaliacao1 = 9.871

const avaliacao2 = 6.871

const total = avaliacao1\*peso1 + avaliacao2\*peso2

const media = total / (peso1+peso2)

console.log(`Média Arredondada: ${media.toFixed(2)}`) /\* Imprimindo com apenas duas casas decimais e formatando a string para a variável ser inserida dentro \*/

console.log(`Média Em String: ${media.toString()}`) /\* Converte a média para tipo string \*/

console.log(`Média Em formato Binário: ${media.toString(2)}`) /\* Converte para valor binário \*/

console.log(`Tipo: ${typeof media}`)

console.log(typeof Number)

## Number: Alguns Cuidados

console.log(7/0) // Infinity

console.log('10'/ 2) // 5 - Identifica um número dentro da string e realiza a divisão

console.log(‘3’ + 2) // 32 - Como é ‘+’, soma em uma string

console.log(‘3’ - 2) // 1 - Subtrai aritmeticamente.

console.log('Show' \* 2) // Retorna um tipo vazio NaN

console.log(0.1 + 0.7) /\* Retorna 0.7999999999999999 devido a especificação utilizada no JS, que é menos precisa, porém mais rápida \*/

// console.log(10.toString()) Não converter inteiros em Strings

console.log((10.258148).toFixed(2))

## Usando o Math

const raio = 5.6 // cm

const area = Math.PI \* Math.pow(raio,2)

console.log(`Área: ${area.toFixed(2)} cm`) // Área: 98.52 cm

console.log(`Math é do tipo: ${typeof Math}`) //Math é do tipo: object

## Tipos em JavaScript: Strings

Strings podem ser delimitados por aspas simples ‘string’, aspas duplas “string”

const escola = "Cod3r"

console.log(`Posição 4: ${escola.charAt(4)}`) // Retorna o caracter na posição 4

console.log(`Posição Inexistente: ${escola.charAt(6)}`) // Como não existe a posição 6, retorna vazio

console.log(escola.charCodeAt(3)) // Retorna o Valor da variável na posição 3 dentro da tabela ASCII

console.log(escola.indexOf('3')) // Retorna o indice do caracter '3' dentro da string escola

console.log(escola.substring(1)) // Fatia a string, da posição 1 para frente

console.log(escola.substring(0, 3)) // Imprime do indice 0 ao 3, sem incluir o 3

console.log('Escola: '.concat(escola).concat("!")) // Concatenar strings

console.log(escola.replace(3, 'e')) // Substitui o digíto 3 pela letra e

console.log(escola.replace(/\d/, 'e')) // Substitui todos os dígitos(/\d/) pela letra e

console.log(escola.replace(/\w/, 'e')) // Substitui o primeiro digito(/\w/) pela letra e

console.log(escola.replace(/\w/g, 'e')) // Substitui todos os dígitos (/\w/g) pela letra e (g = global)

console.log('Ana,Maria,Pedro'.split(',')) // Splita por ',' a string e preenche dentro de um array.

// charAt() - Loop Através de uma String

for (let index = 0; index < escola.length; index++) {

console.log(escola.charAt(index))

}

// Loop Através de um array

var a = ["a", "b", "c"];

for (index = 0; index < a.length; index++) {

console.log(a[index])

}

// Popular um array

var a = ["a", "b", "c"]

array = []

for (index = 0; index < 20; index++) {

array[index] = index\*2

}

## Templates Strings

// Sem usar Template Strings

const nome = 'Liana'

const concatenacao = 'Eu amo a ' + nome + '!'

console.log(concatenacao)

// Usando Template Strings

const nome\_ = 'Liana'

const concatenacao\_ = `Eu amo a

${nome\_}!`

console.log(concatenacao\_)

console.log(`1 + 2 = ${1+2}`)

const up = texto => texto.toUpperCase() // Definição da Função 'up'

console.log(`Ei.... ${up('cuidado')}`) // usando a Função

console.log(`Ei.... ${'cuidado'.toUpperCase()}`) // Usando o método

## Tipos em JavaScript: Booleanos

let isAtivo = false

console.log(isAtivo)

isAtivo = true

console.log(isAtivo)

isAtivo = 1 // 1 Para VERDADEIRO

console.log(!isAtivo) /\* ! Representa o Inverso, como isAtivo é 1, ou seja, true, vai retornar false \*/

console.log(!!isAtivo) /\* !! Representa o Inverso do Inverso, como isAtivo é 1, ou seja, true, vai retornar true \*/

console.log(!true) // Inverso de true é false

// Todos os números inteiros são verdadeiros com exceção do 0

console.log('os verdadeiros...')

console.log(!!3)

console.log(!!-1)

console.log(!!' ')

console.log(!![]) // array

console.log(!!{}) // object

console.log(!!Infinity)

console.log(!!(isAtivo = true)) /\* Pega o valor da atribuição, ou seja, true, e retorna esse valor, no caso true. Não pega se a atribução deu ou não certo \*/

console.log(!!(isAtivo = 23))

console.log('\n')

// Todos os falsos

console.log('os falsos...')

console.log(!!0)

console.log(!!'') /\* String vazia, para testes se um nome, por exemplo, está vazio \*/

console.log(!!null)

console.log(!!NaN)

console.log(!!undefined)

console.log(!!(isAtivo = false))

console.log('\n')

// Resumo

console.log('para finalizar...')

console.log(('' || null || 0 || 'teste' || 13)) /\* || - (Expressão OU). Retorna o primeiro valor com atribuição verdadeiro, que é a string 'teste' \*/

let nome = ''

console.log(nome || 'Desconhecido') /\* "Se nome não estiver preenchido, imprime desconhecido. nome ||(OU) 'Desconhecido', o primeiro true \*/

## Tipos em JavaScript: Array

Array permite agrupar múltiplos valores de forma unidimensional, através de uma estrutura só, com um único identificador.

Em JavaScript os arrays são:

* Heterogêneos: Aceita diferentes elementos de diferentes tipos;
* Tamanho dinâmico: Pode assumir diferentes tamanhos ao longo do algoritmo.

const valores = [1.1,2.2,3.3,4.4,5.5]

console.log(valores[0], valores[3])

console.log(valores[5]) // Retorne undefined para posições inexistentes

valores[10] = 9.9 /\* Atribui o valor 10 para a posição 10, como temos apenas 4 posições, 5 ficarão vazias \*/

console.log(valores[10])

console.log(valores.length) // Tamanho do array

valores.push(6.6) // Para adicionar um valor ao array

valores.push({id:3}, false, null, NaN, 'teste')

console.log(valores)

console.log(valores.pop()) /\* Retira o último valor do array e retorna o valor retirado \*/

delete valores[0] // Deleta o valor da posição 0

console.log(valores)

console.log(typeof valores) // Em JS arrays são do tipo object

## Tipos em JavaScript: Object

JavaScript é uma linguagem multiparadigmas, podemos programar procedural, OO, funcional, etc...

Um objeto em JS é diferente de um JSON (JavaScript Object Notation), que é um formato textual de objetos, mas não é um objeto.

const pessoa1 = {}

pessoa1.nome = 'Liana'

pessoa1.idade = 27

pessoa1['Altura'] = 1.65 /\* Outra forma de atribuir um parâmetro para um objeto (Evitar Atributos com espaço) \*/

console.log('Pessoa 1: ')

console.log(pessoa1)

const pessoa2 = {

nome: 'Lucas',

idade: 29,

Altura: 1.72

}

console.log('Pessoa 2: ')

console.log(pessoa2)

## Entendendo o Null & Undefined

Atribuição por Valor x Atribuição por Referência

const a = {name: ‘Teste’}

const b = a

Ao definir a constante a = {name: ‘Teste’}, ela não tem o conteúdo desse objeto, a constante a tem o endereço apontando para onde o objeto está localizado.

Ao realizar const b = a, dizemos que b, também vai ter o mesmo endereço de a, ou seja, vai apontar para o mesmo local.

Dessa forma, ao alterarmos alguma propriedade de b, vamos alterar a propriedade do objeto, e as constantes a e b apontam para esse objeto, logo:

b.name = ‘Opa’

a.name = ‘Opa’ // Também é alterado.

Esse tipo de atribuição é por referência.

Essa atribuição não ocorre quando trabalhamos com valores primitivos, por exemplo numbers.

let c = 3

let d = 4

d++

d = 4

c = 3

Aqui, a atribuição é por valor, ou seja, d recebe o mesmo valor de c, e ambos apontam para diferentes endereços, de forma que cada um vai possuir um valor, que ao ser alterado individualmente, não altera o outro.

let valor // não inicializada

console.log(valor) // undefined Declarada porem sem valor

//console.log(valor2) // Não declarada

valor = null /\* Variáveis null não tem valor e não está apontando para nenhum endereço de memória, porem está definida \*/

console.log(valor)

console.log(valor.toString()) // Erro de tipo, não podemos usar metodos de variáveis null

const produto = {}

console.log(produto.preco) // Retorna undefined, pois produto foi definida, mas o preco não

//console.log(produto.preco.a) // Retorna Erro, pois não podemos acessar 'a' de algo indefinido, que é o preco

produto.preco = 3.5

console.log(produto)

produto.preco = undefined // EVITAR- Se quiser retirar um atribudo de um objeto, use o delete produto.preco

console.log(!!produto.preco) // Convertendo o valor para boolean

console.log(produto)

produto.preco = null // Sem preço. Não é preco zero, é preco não definido.

console.log(produto)

console.log(!!produto.preco) // Tambem retorna false, pois o preco não está setado.

/\*

Variávels null, sem valor, apontando para nenhum endereço,

não podem ser acessadas

\*/

## Quase Tudo é Função

Objects e Class são functions em JavaScript

console.log(typeof Object)

class Produto {}

console.log(typeof Produto)

## Exemplos Básicos de Funções #01

// Função sem retorno

function imprimirSoma(a, b){

console.log(a + b)

}

imprimirSoma(2,4) // Retorna a soma

imprimirSoma(2) // Se passar apenas 1 parâmetro, o segunda é tido como undefined, e é retornado NaN

imprimirSoma(2,4,3,7,8,9) // Atribui os dois primeiros valores para a e b respectivamente e ignora o restante

imprimirSoma() // Retorna NaN

// Função com retorno

function soma(a, b = 1){

soma\_ = a + b

return soma\_

}

console.log(soma(3)) // Agora passando apenas 1 valor, retorna soma pois b foi definida como '0' caso não passe valor

## Exemplos Básicos de Funções #02

// Armazenando uma função em uma variavel

const imprimirSoma = function (a, b) {

console.log(a+b)

}

imprimirSoma(2,5)

// Armazenando uma funcao arrow em uma variavel

const soma = (a,b) => {

return a + b

}

console.log(soma(3,7))

// Retorno implícito

const subtracao = (a,b) => a-b

console.log(subtracao(10,2))

## Declaração de Variáveis com Var #01

Escopo de um código: É um determinado local onde uma variável é visível

Variáveis declaradas dentro de algum bloco, com var, podem ser acessadas fora do bloco.

Variáveis declaradas dentro de um bloco de uma função só podem ser acessadas dentro da função.

Variáveis criadas fora de funções são variáveis globais, estará disponível em todo objeto de uma página, podendo ser acessada com o window. O Objeto global window só tem no navegador, no node é global

var a = 123

window.a = 123

O Ideal é criar a menor quantidade possível de variáveis no escopo global, para evitar problemas de sobrescrita de variáveis.

## Declaração de Variáveis com Var #02

var numero = 1

{

var numero = 2 // Como a variável é tipo var, vai sobrescrever a declaração anterior

console.log('dentro = ', numero)

}

console.log('fora = ', numero)

## Declaração de Variáveis com let

O let vai manter o escopo dentro do bloco, mesmo o bloco não sendo de uma função

var numero = 1

{

let numero = 2 // Como a variável é tipo let, não sobrescrever a global

console.log('dentro = ', numero)

}

console.log('fora = ', numero)

Quando pedimos para acessar uma variável, o JS vai verificar se a mesma tem no escopo local (Ex: Dentro de uma função), se encontrar usa ela, se não, vai buscar no escopo global.

Var: Escopo global e escopo de função

Let: Escopo global, escopo de função e escopo de bloco

O tipo let foi criado justamente para incluir o terceiro escopo, de bloco. Foi criada, pois caso mudasse o comportamento do var, poderia quebrar diversos sistemas.

## Usando Var em Loop

for (var i = 0; i <10; i++) {

console.log(i)

}

console.log(`i = ${i}`) /\* Como nossa variável i é tipo var, com seguimos obte-la fora do bloco, pois ela não tem escopo de bloco \*/

#Outro Código

const funcs = []

for (var i=0; i<10; i++){

funcs.push(function(){

console.log(i)

})

}

funcs[2]()

funcs[8]()

/\* Apesar das funções na posição 2 e 8, terem sido armazenadas em momentos onde i era diferente

o resultado da impressão da chamada de qualquer função dentro do array funcs vai ser 10, pois é

o atual valor de i. Isso ocorrer pois a variável i é do tipo var, que não tem escopo de bloco

\*/

## Usando Let em Loop

for (var i = 0; i <10; i++) {

console.log(i)

}

console.log(`i = ${i}`) /\* Como nossa variável i é tipo var, com seguimos obte-la fora do bloco, pois ela não tem escopo de bloco \*/

#Outro Código

const funcs = []

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

funcs.push(function(){

console.log(i)

})

}

funcs[2]()

funcs[8]()

/\* Agora como i é variável let, tem escopo de bloco, ou seja, para cada função do array, vai ser armazenado um valor de i 'local', ou seja, cada função vai

imprimir o valor armazenada de i no momento de criação da função

\*/

/\* Em JavaScript funções tem 'consciência do local que foram definidas \*/

## Entendendo Hoisting

/\* Quando declaramos uma variável var, ocorre o hoisting, que é

o comportamento padrão de mover a declaração de variáveis para o topo.

O interpretador da linguagem joga ela para cima

\*/

console.log('a =', a)

var a = 2

console.log('a =', a)

/\* Ao imprimir a variável a na linha 5, antes defini-la

não temos um erro, ela é entendi-da como 'undefined', pois o interpretador

entende que ela foi vai ser declarada abaixa

\*/

/\* Dentro de função \*/

function teste(){

console.log('a =', a)

var a = 2

console.log('a =', a)

}

teste()

/\* Com let \*/

function teste2(){

console.log('a =', a)

let a = 2

console.log('a =', a)

}

// teste2() ERRRO

## Função Vs Objeto

console.log(typeof Object) /\* - Type: function \*/

console.log(typeof new Object) /\* Agora estamos instanciando um objeto - Type: Object \*/

/\* new Object é semelhante a chamada de um construtor \*/

/\* Funções tambem são tipo function, a menos que instancia-mos com new \*/

const Cliente = function(){}

console.log(typeof Cliente) /\*Criando instancias de funções que definimos \*/

console.log(typeof new Cliente)

/\* Para classes o comportamento é o mesmo. Classes são formas diferente de construir funções.

Internamente é criado uma função \*/

class Produto {} // Padrão ES 2015 (ES6) - Criar instancia a partir de classes

console.log(typeof Produto)

console.log(typeof new Produto)

//OBS: Classes e Funções podem ser chamadas com parênteses Cliente(), Produto()

/\*

Quando criamos classes a partir da notação literal, ja temos a instancia

Que é diferente de definir uma função, que é como se fosse a sua classe (molde)

e a partir dessa estrutura, você instancia quantos objetos quiser.

\*/

// Em JS, quem exerce o papel de criar classes e instanciar objetos é a função

## Par Nome/Valor

/\* Um Objeto é um conjunto chave/valor \*/

const saudacao = 'Opa' /\* saudacao=chave 'Opa'=valor (Podemos usar o conceita para variáveis tambem)

Nossa variável saudacao foi definida dentro de um contexto lexico (Local onde a variável foi definida fisicamente)

, no caso lexico 1.

Contexto lexico 1 - Contexto local

Dentro de um código temos vários contextos, uma função por exemplo define um contexto (contexto local da função)

Se tentarmos acessar uma variável/constante dentro de contexto de uma função e não encontrar, internamente

essa variável/constante é procurada em contextos mais abrangentes.

\*/

function exec(){

const saudacao = 'Hey' // Par chave valor / contexto léxico 2

return saudacao

}

// a constante saudacao foi definida 2x, porem em contextos diferentes, não ocorre conflito.

// Objetos são grupos aninhados de pares chave/valor

const cliente = {

nome: 'Pedro',

idade: 26,

peso: 140,

altura: 1.50,

endereco: {

logradouro: 'Rua dos bobos', // logradouro está em contexto

numero: 0

}

}

console.log(saudacao)

console.log(exec())

console.log(cliente)

## Notação Ponto

console.log(typeof console) // console em JS é um objeto

console.log(Math.ceil(6.1)) // ceil arredonta para cima 7

console.log(Math.floor(6.1)) // ceil arredonta para baixo 6

// Math tambem é um objeto em JS

const obj1 = {} //Objeto criado sem atributos

obj1.nome = 'Liana' // Notação ponto para criar dinâmicamente um objeto obj1['nome'] = 'Liana' NÂO USAR

obj1.personalidade = 'Brava'

obj1.beleza = 'Absurdo'

obj1.status = 'Amor da Minha vida'

function obj(nome) {

this.nome = nome

/\* this.nome que dizer que vamos receber o atributo nome por parâmetro

e esse parâmetro ficará visivel para quem criar um objeto do tipo Obj

Ou seja, um atributo publico \*/

this.exec = function (){ /\* Podemos também passar funções como parâmetros de objetos \*/

console.log("Exec....")

}

}

const obj2 = new obj('Liana') //Instanciando um objeto obj 2 a partir de obj

const obj3 = new obj('Lucas')

console.log(obj2.nome)

console.log(obj3.nome)

obj2.exec()

obj3.exec()

## Operadores: Atribuição

const a = 7

let b = 3

b+=a // Acrescentando em b, a

console.log(b)

b-=4 // b = b - 4

console.log(b)

b\*=2 // b = b \* 2

console.log(b)

b/=2 // b = b / 2

console.log(b)

b%=2 // b = b % 2 Resto da divisão

console.log(b)

## Operadores: Destructing #01

/\* É um operador para destruturação:

\* Retirar atributos de um objeto

\* Retirar elementos de um array

\* Forma simplificada para obter elementos de uma estrutura

\*/

const pessoa = {

nome: 'Liana',

idade: 27,

endereco: {

e1: 'Campinas',

e2: 'Bauru'

}

}

const { nome, idade } = pessoa /\* tire ({ }) do objeto (= pessoa) os atributos nome e idade

Tira do objeto os atributos e atribui as respectivas variáveis nome e idade, mesma

nomenclatura que tinham dentro do objeto \*/

console.log(`Nome: ${nome}\nIdade: ${idade}`)

//Podemos colocar tambem os parâmetros em variáveis com outros nomes (n, i)

const { nome: n, idade: i } = pessoa

console.log(`Nome: ${n}\nIdade: ${i}`)

//Se as variáveis que passamos no destructing, não existirem no objeto, retorna undefined

const { sobrenome, bemHumorada = true } = pessoa // sobrenome e bemHumorada não existem, retorna undefined e true

console.log(sobrenome, bemHumorada)

//Para obter elementos de paramêtros que possuem parametros, como endereco

const { endereco: { e1, e2, e3 } } = pessoa /\*

Não obterm todos os dados de endereco, apenas os que passamos (cria e1, e2 e e3, não endereco) \*/

console.log(e1, e2, e3)

const { conta: { num, ag } } = pessoa // Teremos erro, pois pessoa não tem o parametro conta, muito menos num e ag

## Operadores: Destructing #02

const [a] = [10] // a = 10

const [n1, , n3, , n5, n6 = 0] = [1, 2, 3, 4]

console.log(n1, n3, n5, n6)

const [, [, nota]] = [[, 8, 8], [9, 6, 8]]

console.log(nota)

## Operadores: Destructing #03

function rand({ min = 0, max = 1000 }) { // Aqui estamos passando o operador destructing

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

return Math.floor(valor)

}

const obj = { max: 50, min: 40 }

console.log(rand(obj)) // Aqui estamos passando um objeto

console.log(rand({min:955})) // Passando com parâmetro

console.log(rand({})) // Passando um objeto vazio - Vai pegar o min e max da função (0 e 1000)

console.log(rand()) // Aqui teremos erro pois vai tentar desestruturar algo nulo ou undefined

## Operadores: Destructing #04

function rand([min = 0, max = 1000]) { //Quando destruturamos um array, não recebemos um array e sim os atributos, no caso 2

if (min > max) [min, max] = [max, min]

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

return Math.floor(valor)

}

console.log(rand([50,40]))

console.log(rand([999]))

console.log(rand([, 10]))

console.log(rand([]))

//console.log(rand()) /\* Aqui teremos erro pois vai tentar desestruturar algo nulo ou undefined \*/

/\* Resumo

O Operador Destructing é uma forma de obter elementos de objetos e arrays

para criar variáveis de uma vez só, uma forma simples de criar ou extrair de

elementos aquilo que precisamos. Sabemos que vamos receber um objeto, por ex-

emplo, a partir de uma função, ja podemos, em vez de trabalhar com o objeto

dentro da função, tirar os atributos que queremos e operamos de uma forma bem

mais direta, sem precisar acessar o objeto.

\*/

## Operadores: Aritméticos

Os operadores possuem algumas formas de classificação, como tipo: Aritméticos, relacionais, lógicos, de atribuição, etc... ou quanto a quantidade de operandos, os aritméticos possuem 2 operandos (binário).

const [a, b, c, d] = [3, 5, 1, 15]

const soma = a + b + c + d // Operações binárias --> a+b | (a+b)+c | (a+b+c)+d Operadores infix

const subtracao = d - b

const multiplicacao = a \* b

const divisao = d / a

const modulo = a % 2

console.log(`Soma: ${soma}`)

console.log(`Subtração: ${subtracao}`)

console.log(`Multiplicação: ${multiplicacao}`)

console.log(`Divisão: ${divisao}`)

console.log(`Modulo: ${modulo}`)

// d++ //Operador postfix - Depois do operando

## Operadores: Relacionais

Em uma operação relacional, o resultado sempre será ‘true’ ou ‘false’

É possível comprar se valores são iguais (mesmo valor), ou se são estritamente iguais (mesmo valor e tipo).

console.log('01)', '1' == 1) // '1' é igual 1 - true, interpreta o valor numérico

console.log('02)', '1' === 1) // '1' é igual 1 - false, pois '1' tem o mesmo valor numérico que 1, mas não é estritamente igual (string x number)

console.log('03)', '3' != 3)

console.log('04)', '3' !== 3)

console.log('05)', 3 < 2)

console.log('06)', 3 > 2)

console.log('07)', 3 <= 2)

console.log('08)', 3 >= 2)

const d1 = new Date(0) // 0 - 01/01/1970

const d2 = new Date(0)

console.log('09)', d1 == d2) // Vai comparar endereços de memória, falso, pois cada constante tem o seu

console.log('10)', d1 === d2) // Também vai comparar endereços de memória, falso, pois cada constante tem o seu

/\* Trabalhando com variáveis de endereço de memória (d1 e d2)

não faz diferença, sempre vai dar false tanto para '==' quanto '===' \*/

console.log('11)', d1.getTime() == d2.getTime())

console.log('12)', d1.getTime() === d2.getTime()) // Ambos são estritamente iguais/

console.log('13)', undefined == null) // true, são iguais

console.log('14)', undefined === null) /\* false, não são estritamente iguais, tipos diferentes \*/

## Operadores: Lógicos

/\*

\* v e v -> v

\* v e f -> f

\* f e ? -> f

\* v ou ? -> v

\* f ou v -> v

\* f ou f -> f

\* v xor v -> f xor : Exclusivo ou

\* v xor f -> v

\* f xor v -> v

\* f xor f -> f

\* !v -> f Contrário de v

\* !f -> v Contrário de f

\*/

/\* t1 = Trabalho Terça | t2 = Trabalho Quinta

Se t1 e t2 der certo ---> Compra TV 50" e Tomar Sorvete

Se t1 ou t2 der certo ---> Compra TV 32" e Tomar Sorvete

Se t1 e t2 der errado ---> Não Compra TV nem toma Sorvete

Para comprar a TV 50", precisamos de um operador 'E', pois só podemos

compra-la se t1 e t2 forem true, para Tomar Sorvete, precisamos do 'OU',

pois se t1 ou t2 forem true, tomamos sorvete. Para Comprar a TV 32",

precisamos do operador xor (Exclusivo), pois compra

\*/

function compras(t1, t2) {

const comprarSorvete = t1 || t2 // || = Ou

const comprarTV50 = t1 && t2 // && = E

// Opção 1: const comprarTV32 = !!(t1 ^ t2) //operador bitwise xor

const comprarTV32 = t1 != t2 // O 'Ou Exclusivo' pode ser simulado com uma diferença.

const manterSaudavel = !comprarSorvete // manterSaudavel é True quando comprarSorvete é falso.

return { comprarSorvete, comprarTV50, comprarTV32, manterSaudavel } /\*

Objetos só com valor ? Quando omitimos a chave, passando uma constante ou variável

para criação do objeto, é um recurso novo do ES 2015. Quando criamos dessa forma, a chave

vai ser a constante e o valor vai ser o valor da constante (true ou false)

\*/

}

console.log(compras(true, true))

console.log(compras(true, false))

console.log(compras(false, true))

console.log(compras(false, false))

## Operadores: Unário

let num1 = 1

let num2 = 2

num1++ // Pós fixado

console.log(num1)

--num1 //Pré fixado

console.log(num1)

/\* Assim como multiplicação tem prioridade em relação a soma,

operações pré fixadas tem prioridade em relação as pós fixadas. \*/

console.log(++num1 === num2--) // true ou false ?

/\*

Lógica é incrementado 1 em num1, que vale 2, que é comparado com num2

que vale 2, e só depois, é 'decrementado' de num2.

Deixe o código mais simples possível.

\*/

## Operadores: Ternário

// Operador Ternário: Três Operandos

const resultado = nota => nota >=7 ? 'Aprovado': 'Reprovado'

/\* Essa função arrow (função resultado), retorna implicitamente uma das opções

passadas para o operador ternário. \*/

/\* O Operador ternário é composto por algumas partes

\* Expressão lógica: nota >=7 (Que retorna true ou false)

\* A interrogação (?)

\* Retorno caso a expressão seja falsa ('Aprovado')

\* Dois pontos (:)

\* Retorno caso a expressão seja true ('Reprovado)

\*/

// Operador: nota >=7 ? 'Aprovado': 'Reprovado'

console.log(resultado(8.0))

/\* Outra forma

const resultado = nota => {

return nota>=7 ? 'Aprovado': 'Reprovado'

}

console.log(resultado(8.0))

\*/

## Contexto de Execução: Browser x Node

Runtimes para executar JS

* FrontEnd: Executado dentro do browser
* BackEnd: Executado dentro do node

Executando no browser, temos o objeto window, com diversas propriedades, definindo o escopo mais abrangente de qualquer script.

O Objeto this, pode variar, mas se não for passado essa variação, ele é idêntico ao window.

Console:

var a = 123

window.a = 123

let b = 123

window.b = undefined // Variáveis let não vão para o escopo global window/this, mas são globais, podendo ser acessadas pelo browser.

OBS: Constantes e variáveis let não vão para o objeto window.

// NÃO COMPARTILHAR VARIÁVÉIS DE FORMA GLOBAIS

Funções criadas são armazenadas no window, podendo ser acessadas:

function f1(){ return this === window }

window.f1() : true

this.f1(): true

# Estruturas de Controle

## Usando a Estrutura IF #01

function soBoaNoticia(nota) {

if(nota >= 7){

console.log('Aprovado com ' + nota)

}

}

soBoaNoticia(8.1)

soBoaNoticia(6)

function seForVerdadeEuFalo(valor){

if(valor){

console.log('É verdade... ' + valor)

}

}

seForVerdadeEuFalo() // Passa undefined, que é false, não cai no if

seForVerdadeEuFalo(null) //falso

seForVerdadeEuFalo(undefined) //falso

seForVerdadeEuFalo(NaN) //falso

seForVerdadeEuFalo('') //falso

seForVerdadeEuFalo(0) //falso

seForVerdadeEuFalo(-1) //verdadeiro

seForVerdadeEuFalo(' ') //verdadeiro

seForVerdadeEuFalo('?') //verdadeiro

seForVerdadeEuFalo([]) //verdadeiro

seForVerdadeEuFalo([1,2]) //verdadeiro

seForVerdadeEuFalo({}) //verdadeiro

## Usando a Estrutura IF #02

function teste1(num) {

if (num > 7)

console.log(num) // Sentença associada ao if

console.log('final') // Sentença não associada ao if (Sempre será executada)

}

teste1(6) /\* Não cai no if, imprime apenas 'final', que sempre será executada,

por nosso if n tem aspas, ou seja, está associado apenas a primeira sentença console.log(num),

a segunda em diante, sempre será executada \*/

teste1(8) /\* Em JS as 'aspas' para definir um bloco if, não é obrigatório, porem,

apenas uma sentença estará associada aquele bloco \*/

function teste2(num) {

if (num > 7); {

console.log(num)

}

}

teste2(6) // Apesar de ser menor que 7 é impresso, pois o ';' encerra o bloco if; não considera o bloco {}

teste2(8)

## Usando a Estrutura IF/ELSE

function tratarErroELancar(erro) {

throw console.log('Digite um númer')

}

const imprimirResultado = function (nota) {

// Is Not a Number

if (!isNaN(nota) == false) {

console.log('Digitar Nota númerica')

return

}

if (nota >= 7) {

console.log('Aprovado!')

} else {

console.log('Reprovado!')

}

}

imprimirResultado(10)

imprimirResultado('Epa') //por JS ser fracamente tipada, como 'Epa' não é > 7, cai no bloco else

var valor = 1.5;

var teste = "a";

console.log(!isNaN(valor)); /\* retorna false negando fica true, 1.5 é tipo numérico \*/

console.log(!isNaN(teste)); /\* retorna true negando fica falso, "a" não é um número \*/

console.log(!isNaN('10'))

## Estrutura IF/ELSE IF

/\* Função associada a um número. Retorna true se um número

está entre os parâmetros e false se não está \*/

Number.prototype.entre = function (inicio, fim) {

return this >= inicio && this <= fim

// Se é Number, acessamos com this

}

const imprimirResultado = function (nota) {

if (nota.entre(9, 10)) {

console.log('Quadro de honra')

} else if (nota.entre(7, 8.99)) {

console.log('Aprovado')

} else if (nota.entre(4, 6.99)) {

console.log('Exame')

} else if (nota.entre(0, 3.99)) {

console.log('Bombou')

} else {

console.log('Nota Inválida')

}

console.log('Fim')

}

imprimirResultado(7)

## Usando a Estrutura Switch

Switch é uma estrutura para trabalhar com múltiplas seleções

/\* O Switch é feito para múltiplas seleções.

A expressão do switch não retorna true ou false,

retorna um valor, que pode ser alguns tipos

Uma vez que o valor caiu em um determinado case, ele sai

executando todos os outros abaixo, para sair, vamos precisar

colocar break \*/

const imprimirResultado = function (nota) {

switch (Math.floor(nota)) {

case 10:

case 9:

console.log('Quadro de honra') // Entra nessa linha para 10 e 9

break

case 8: case 7:

console.log('Aprovado')

break

case 6: case 5: case 5:

console.log('Exame')

break

case 3: case 2: case 1:

console.log('Reprovado')

break

default:

console.log('Nota Inválida')

}

}

imprimirResultado(-1.2)

## Estrutura While

Par situações onde temos uma quantidade indeterminada de repetições

function getInt(min, max) {

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

return Math.floor(valor)

}

let opcao = 0

while (opcao != -1){

opcao = getInt(-1,10)

console.log(`Opção escolhida foi: ${opcao}`)

}

console.log('fim')

## Estrutura FOR/IN

/\* For IN percorre os atributos de determianda estrutura (Objetos e Arrays)

Existem outras formas melhores de percorrer arrays:

\* For Int

\* map

\* reduce

\* filter

\*/

const notas = [6.7, 7.4, 9.8, 8.1, 7.7]

for (let i in notas) { //Existem formas melhores

console.log(i, notas[i])

}

const pessoa = {

nome: 'Liana',

sobrenome: 'Fernandes',

idade: 29,

peso: 64

}

//Percorrendo um objeto

for (let atributo in pessoa){

console.log(`${atributo} = ${pessoa[atributo]}`)

}

## Usando break/continue

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

for (let x in nums) {

if (x == 5) {

break

}

console.log(`x = ${x}`)

}

//break e continue agem em blocos for, while e switch

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

for (let x in nums) {

if (x == 5) {

continue

}

console.log(`x = ${x}`)

}

// Criando um rótulo associado a um loop for

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

externo: for (let a in nums) { // esse for loop está associado a um laço externo

for (let b in nums) {

if (a == 2 && b == 3) break externo // Interrompe o for associado ao rótulo externo

//break o break interrompe o próximo for, se quisermos interronper outro, podemos fazer com um rótulo (externo)

console.log(`Par = ${a}, ${b}`)

}

}

# Funções

## Cidadão de Primeira Linha

Possibilidades para trabalhar com função em JS

/\* Função em JS é First-Class Object (Citizens)

Higher-order function

Função em JS é tratada com um dado, onde podemos:

Retornar uma função dentro de uma função

Armazenar em uma variável

Armazenar em um array

\*/

//Criar função de forma literal

function fun1() { }

//Armazenar função em uma constante

const fun2 = function () { }

//Armazenar função em um array

const array = [function (a, b) { return a + b }, fun1, fun2]

console.log(array[0](10, 5))

//Armazenar função em atributos de objetos

const obj = {}

obj.falar = function () { return 'Opa' }

console.log(obj.falar())

//Passar função como parâmetro de uma função - Abre muitas possibilidades

function run(fun) {

fun()

}

run(function () { console.log('Executando....') })

//Uma função pode retornar/conter uma função

function soma(a, b) {

return function (c) {

console.log(a + b + c)

}

}

soma(1,2)(4) //soma(1,2) é uma função, que recebe 4 como parâmetro

//Outra forma

const fun = soma(1,2)

fun(4)

## Parâmetros e retorno são opcionais

function area(l, h) {

const area = l \* h

if (area > 20) {

console.log(`Valor acima do permitido: ${area} m2.`)

} else {

return area

}

}

console.log(area(1, 2))

console.log(area(1)) // 2 \* undefined = NaN

console.log(area(1, 2, 3, 4))

console.log(area(5, 5)) // Retorna undefined

## Parâmetros Variáveis

// Uma função pode ser declarada sem parÂmetros e através de arguments, podemos recuperar esses parâmetros

function soma(){

let soma = 0

for (i in arguments) {

soma += arguments[i]

}

return soma

}

console.log(soma()) //Sem parâmetros não retorna nada

console.log(soma(1)) //1 parâmetro passado, retorna ele

console.log(soma(1.1, 2.2, 3.3)) //recuperamos 1.1, 2.2 e 3.3 como argumentos através do arguments

## Parâmetro Padrão

// estratégia 1 para gerar valor padrão

function soma1(a, b, c) {

a = a || 1 //se a for true (passado) recebe a, se false(não passado ou zero) retorna 1

b = b || 1

c = c || 1

return a + b + c

}

console.log(soma1()) //Assumiu 1 para todos, como previsto

console.log(soma1(3))

console.log(soma1(1, 2, 3))

console.log(soma1(0, 0, 0)) //O valor 0 retorna falso, assumindo 1

//estratégia 2, 3 e 4 para gerar valor padrão

function soma2(a, b, c) {

a = a !== undefined ? a : 1 //1ª Estratégia

b = 1 in arguments ? b : 1 //2ª Estratégia - b - 1... (1 é o índice do parâmetro 2, indice 1)

c = isNaN(c) ? 1 : c //3ª Estratégia - Melhor Opção (Com excessão da opção nova do ES2015)

return a + b + c

}

console.log(soma2())

console.log(soma2(3))

console.log(soma2(1, 2, 3))

console.log(soma2(0, 0, 0))

// Usando o valor padrão do ES2015 - Forma mais usual

function soma3(a = 1, b = 1, c = 1) {

return a + b + c

}

console.log(soma3())

console.log(soma3(3))

console.log(soma3(1, 2, 3))

console.log(soma3(0, 0, 0))

/\*A Opção do valor padrão serve para parÂmetros passados para funções

Mas não serve para variáveis cridos dentro de funções. Nesse caso usar

estratégia 1 \*/

## ‘this’ pode variar

/\* this, assim como self em python, é uma palavra reservada

que serve para referenciar um objeto atual de uma execução

Em JS o this pode variar conforme a forma que a função é chamada.

O this é sempre o objeto que está sendo referenciado naquele contexto de execução

Objetivos para a criação de arrow functions

\* Simplificação da chamada (Diminuir a sintaxe)

\* this não variar (Ser constante) conforme a chamada (contexto léxico - Aquele no qual a função foi chamada)

Em funções declaradas da forma tradicional function f3() {},

o this pode variar conforme quem chamou a função, a menos que

usemos uma função arrow

\*/

## ‘this’ e a função bind #01

const pessoa = {

saudacao: 'Bom dia!',

falar(){

console.log(this.saudacao) //this aqui se refere ao objeto pessoa

}

}

pessoa.falar()

const falar = pessoa.falar

falar() /\* Vai retornar undefined, pois está apontando um this diferente,

que não é o objeto pessoa, que não tem parâmetro saudacao.

Conflito entre paradigmas: funcional e OO \*/

//Para resolver isso, podemos usar o bind

const falarDePessoa = pessoa.falar.bind(pessoa) /\* Com o bind passamos o objeto que queremos

que seja resolvido o this

Aqui o this vai ser o objeto que definimos na linha 1 (pessoa)

\* O bind é o método responsável por amarrar um determinado objeto, para ele ser dono da execução daquele metodo, sempre que o metodo for chamado.

\* O this, ao chamar a função definida (no caso falarDePessoa), vai se referenciar

ao objeto passado como parâmetro no bind

\* A função bind retorna outra função, que nessa função está amarrado o objeto pessoa, e essa função está armazenada na constan falarDePessoa\*/

falarDePessoa()

## ‘this’ e a função bind #02

//Ja que o this varia, como cravar seu valor ?

function Pessoa(){

this.idade = 0

setInterval(function () {

this.idade++

console.log(this.idade)

}.bind(this), 1000) //Dispara outra função a cada 1000 ms

}

/\* Sem o bind, o this não se refere ao objeto pessoa, pois foi definido dentro da

função setInterval \*/

new Pessoa //instanciando o objeto Pessoa

/\* Outra forma: Armazenar o this em uma constante no momento em que queremos referenciar,

para acessar o this em outro momento

\*/

function Pessoa(){

this.idade = 0

const self = this //self não muda nunca - Forma de driblar o fato do this mudar

//com essa forma (definindo a constante self, não importa quem chama a função rederenciando o this)

//Podemos assim acessar com a constante o que o this acessaria naquele momento

setInterval(function () {

self.idade++

console.log(self.idade)

}, 1000) //Dispara outra função a cada 1000 ms

}

/\* Sem o bind, o this não se refere ao objeto pessoa, pois foi definido dentro da

função setInterval \*/

new Pessoa //instanciando o objeto Pessoa

## Funções Arrow #01

let dobro = function (a) {

return 2 \* a

}

console.log(dobro(Math.PI))

//Escrevendo a mesma função como arrow

dobro = (a) => { return 2 \* a }

console.log(dobro(Math.PI))

//Forma reduzida, para 1 parâmetro - return implicito

dobro = a => 2 \* a

//As funções arrow são anônimas, para chamar temos que armazenar em constantes

console.log(dobro(Math.PI))

//Função sem parâmetro

let ola = function (){

return 'Olá!'

}

ola = () => 'Olá!'

ola = \_ => 'Olá!' //possui parâmetro (\_), mas pode ser ignorado

console.log(ola())

## Funções Arrow #02

function Pessoa(){

this.idade = 0

setInterval(() => {

this.idade++

console.log(this.idade)

}, 1000)

}

new Pessoa

/\* Aqui, não é necessário usar bind nem o const self = this, pode definimos nossa função

como uma função arrow, ou seja, o this não varia, aponta para o objeto do contexto da função \*/

## Funções Arrow #03

let comparaComthis = function (param) {

console.log(this == param)

}

comparaComthis(global) //Aqui o this aponta para o objeto global

const obj = {}

comparaComthis = comparaComthis.bind(obj)

comparaComthis(global) //Agora aponta para o objeto pois pedimos para fazer isso com o bind

comparaComthis(obj)

//Testando esse comportamento com arrow functions

let comparacomThisArrow = param => console.log(this == param)

comparacomThisArrow(global) // Como a função é arrow, não aponta para o global, mas sim para o contexto do módulo (arquivo que foi definida a função)

comparacomThisArrow(module.exports) //Módulo local (arquivo .js em questão)

//Usando o bind dentro de uma função arrow para ver se conseguimos mudar a referência do this

comparacomThisArrow = comparacomThisArrow.bind(obj)

comparacomThisArrow(obj) // false, não aponta

comparacomThisArrow(module.exports) // continue mudando para o contexto que a função foi escrita

## Funções Anônimas

const soma = function (x, y) { //Sem nome = Anônima

return x + y

}

const imprimirResultado = function (a, b, operacao = soma) { //Se não passar o parâmetro operacao, usa a função soma

console.log(operacao(a, b))

}

imprimirResultado(3, 4)

imprimirResultado(3, 4, soma)

imprimirResultado(3, 4, function (x, y) { //Função sem passada como parâmetro

return x - y

})

imprimirResultado(3, 4, (x, y) => x \* y) //Passando uma arro function, que sempre é anônima

const pessoa = {

falar: function () { //Função anônima como parâmetro de um objeto

console.log('Opa')

}

}

pessoa.falar()

## Funções Callback #01

/\* A ideia do callback é passar uma função para outra função, e quando determinado evento acontercer

essa função ser chamada \*/

const fabricantes = ['Mercedes', 'Audi', 'BMW']

function imprimir(nome, indice) {

console.log(`${indice + 1}. ${nome}`)

}

fabricantes.forEach(imprimir) // do array fabricantes, para cada (forEach) item, execute a função

fabricantes.forEach(function (a) {

console.log(a)

})

fabricantes.forEach(fabricante => console.log(fabricante)) //Mesma ideia usando arrow function

//Fizemos um loop no array, cada loop foi um evento, que ao ocorrer, chamou uma função

## Funções Callback #02

const notas = [7.7, 6.5, 5.4, 8.9, 3.6, 4.4, 9.0]

/\* Objetivo:

Gerar um novo array com as notas menores que 7 \*/

//Opção 1 - Sem Callback

const notasbaixas1 = []

for (let i in notas) {

if (notas[i] < 7) {

notasbaixas1.push(notas[i])

}

}

console.log(notasbaixas1)

//Opção 2 - Com Callback

const notasbaixas2 = notas.filter(function (nota) {

return nota < 7

})

console.log(notasbaixas2)

/\* Ao usarmos filter, estamos solicitando que, para o array notasbaixa, passe

apenas as notas no momento do loop que a função passada no filter, for true \*/

//Opção 3 - Com callback e arrow

const notasbaixas3 = notas.filter(nota => nota < 7)

console.log(notasbaixas3)

## Funções Callback #03

//Exemplo de callback no brownser

document.getElementsByTagName('body')[0].onclick = function (e){

console.log('O evento ocorreu')

}

## Funções construtoras

Em JS classes são apenas forma simplificadas de escrever funções.

/\* A função Carro é uma função construtora, que é o molde

dos objetos que podemos criar a partir dessa função

\*/

function Carro(velocidadeMaxima = 200, delta = 5) {

let velocidadeAtual = 0

this.acelerar = function () {

if (velocidadeAtual + delta <= velocidadeMaxima) {

velocidadeAtual += delta

} else {

velocidadeAtual = velocidadeMaxima

}

}

this.getVelocidadeAtual = function () {

return velocidadeAtual

}

}

// Usamos o this em this.acelerar e this.getVelocidadeAtual para tornar publico

const uno = new Carro //Ou new Carro()

uno.acelerar()

console.log(uno.getVelocidadeAtual())

const ferrari = new Carro(350, 20)

ferrari.acelerar()

console.log(ferrari.getVelocidadeAtual())

## Tipos de Declaração

console.log(soma(7, 2))

/\* function declaration - Tem a vantagem de ser lida pelo interpretador

antes da execução, ou seja, podemos chamar a função em uma linha

antes de declaração

\*/

function soma(x, y) {

return x + y

}

/\* function expression - Só podem ser chamadas após a declaração

\*/

const sub = function (x, y) {

return x - y

}

console.log(sub(7, 2))

/\* named function expression - Só podem ser chamadas após a declaração

Pouco usado, vantagem na hora de debugar código,

pois aparece o nome da função

\*/

const mult = function mult(x, y) {

return x \* y

}

console.log(mult(7, 2))

//

## Contexto Léxico

É o contexto no qual as ‘coisas’ foram declaradas dentro da linguagem.

const valor = 'Global'

function minhaFuncao (){

console.log(valor)

}

function exec(){

const valor = 'Local'

minhaFuncao()

}

exec()

/\* A função exec apesar de ter declarado uma constante valor

ao executar a função 'minhaFuncao', vai retornar o valor 'Global'

pois ele foi definida em um contexto léxico (local de declaração)

diferente, ou seja, mesmo chamando ela em outro local, ela carrega

consigo esse contexto, e nesse contexto (funcao 'minhaFuncao') ela

não tinha recebido valor = 'Local', buscou no escopo léxico onde

ela foi definida. Não busca no local de execução, mas sim no local

onde foi definida.

\*/

## Closures

// Closure - Fechamento, algo que envolve algo

// Closure é o escopo criado quando uma função é declarada

/\* Esse escopo permite a função acessar e manipular variáveis

externas à função \*/

const x = 'Global'

function fora() {

const x = 'Local'

function dentro() {

return x

}

return dentro

}

const minhaFuncao = fora() // fora() retorna uma função armazenada em 'minhaFuncao'

console.log(minhaFuncao()) /\* Aqui, vamos retornar a funcao dentro, que foi definida no contexto

léxico da função fora(), dentro do seu escopo, e nesse escopo, tinhas um valor de x igual a 'Local'

ou seja, vamos obter o que ela encontrou no seu contexto \*/

## Função Factory

/\* Função Factory é uma função que retorna um objeto - Uma fábrica de objetos (Sem necessidade de instanciar)

As factorys podem tambem retornar funções agrupadas como parâmetros de um objeto

\*/

function makePeople(nome, sobrenome, idade) {

return {

nome, // não é necessário nome: nome, pois ja passamos nos parâmetros quais são os atributos

sobrenome,

idade

}

}

const pessoaA = makePeople('Lucas', 'Peixoto', 29)

const pessoaB = makePeople('Liana', 'Fernandes', 27)

console.log(pessoaA.nome)

console.log(pessoaA.sobrenome)

console.log(pessoaA.idade)

console.log(pessoaB.nome)

console.log(pessoaB.sobrenome)

console.log(pessoaB.idade)

## Classe vs Função Factory

Classe é uma função escrita de forma diferente

// Classe construtora de Pessoa

class PessoaClass {

constructor(nome) {

this.nome = nome

}

falar() {

console.log(`Meu nome é ${this.nome}`)

}

}

const pessoaA = new PessoaClass('Liana')

pessoaA.falar()

// Função construtora de Pessoa

const PessoaFunction = nome => {

return {

falar: () => console.log(`Meu nome é ${nome}`)

}

}

const pessoaB = PessoaFunction('Liana')

pessoaB.falar()

/\* Na classe, precisamos do this para retornar o nome,

e dependendo do contexto, o this pode variar conforme

quem chamou a função. Esse problema não temos com a

função Factory

\*/

## Immediately Invoked Function Expression (IIFE)

Função auto invocada, que é chamada após a declaração, tudo criado dentro da função auto invocada vai ser escopo local dessa função, tornando assim, mais fácil para fugir do escopo global, impossibilitando que outras partes do sistema acessem as variáveis dentro da função.

// IIFE - Immediately Invoked Function Expression

(function (nome) {

this.nome = nome

console.log(this.nome)

console.log('Será Executado na Hora') //Tudo executado só toca o escopo local

console.log('Foge do Escopo Mais abrangente')

}) ('Lucas')

## Call & Apply

Duas formas diferentes de executar funções

/\* Call e Apply são formas de se chamar funções.

A diferença entre eles está na forma como passamos

os parâmetros \*/

function getPreco(imposto = 0, moeda = 'R$') {

return `${moeda} ${this.preco \* (1 - this.desc) \* (1 + imposto)}`

}

const produto = {

nome: 'Notebook',

preco: 4580,

desc: 0.15,

getPreco

}

global.preco = 20

global.desc = 0.1

console.log(getPreco()) // Usa os valores globais, ja que preco e desc não tem valores padrão

console.log(produto.getPreco())

//Usando call

const carro = { preco: 45000, desc: 0.20 }

console.log(getPreco.call(carro))

console.log(getPreco.call(carro, 0.17, '$'))

//Usando o apply

console.log(getPreco.apply(carro))

console.log(getPreco.apply(carro, [0.17, '$']))

/\*

Quando chamamos o call, passamos nos parametros diretamente os parâmetros necessários

console.log(getPreco.call(carro, 0.17, '$'))

Primeiro o contexto (carro) depois os parÂmetros (0.17 e $)

No apply, os parÂmetos são passados dentro de um array

console.log(getPreco.apply(carro, [0.17, '$']))

\*/

# Objetos & OO

## Introdução à OO

/\*

Antes das linguagens estruturadas, tinhamos o Go To

Procedural

Dentro das linguagens estruturadas, temos os paradigma procedural

que é baseado no procedimento, na função , que realiza um procedi-

mento com um conjunto de dados

processamento(valor1, valor2, valor3)

Orientação à Objetos

Antes tínhamos na procedural funções manipulando dados, agora temos

dados (objetos) e dentro dos dados temos funções.

objeto = {

valor1,

valor2,

valor3,

processamento(){

}

}

objeto.processamento()

O Objeto funciona como uma capa, que agrupa processamentos e atributos.

A OO trouxe uma forma de programar baseado em comportamentos mais humanos,

objetos diferentes tendo seus dados, seus comportamentos e se relacionando

com outros objetos.

Trazer a realizade do mundo para o software.

Principios importantes em OO:

\* Abstração: Pegar um objeto do mundo real e traduzir para dentro do sistema

\* Encapsulamento: Ter os detalhes de implementação escondido e mostrar para

quem precisa saber apenas. Precisamos conhecer a interface de comunicação,

mas não os detalhes de implementação interna. Um objeto deixar exposto todos

os detalhes de sua implementação pode ser algo ruim, pois gerar dependência

desses.

\* Herança: Importante para reuso de código, que um objeto recebe atributos,

características de outros objetos. Em JS não temos herança multipla (Um filho ter 2 pais)apenas.

Se possível optar pela composição (tem um) no lugar da herança apenas.

\* Polimorfismo: Múltiplas formas, temos uma variável do tipo pai, mais genérico (carro), onde

para determinado objeto, podemos atribuir tipos (honda, ferrari, bmw, etc...)

## Revisão sobre Objeto

//Objeto - Coleção dinâmica de pares chave/valor

const produto = new Object

produto.nome = 'Cadeira'

produto.marca = 'Generica'

produto.preco = 220

console.log(produto)

delete produto.preco

console.log(produto)

const carro = {

modelo: 'A4',

valor: 89000,

proprietario: {

nome: 'Liana',

idade: 27,

endereco: {

cidade: 'Bauru',

bairro: 'Centro',

numero: 152

}

},

condutores: [{

nome: 'Lucas',

idade: 29

}, {

nome: 'Raul',

idade: 2

}],

calcularseguro: function(){

return console.log('Seguro...')

}

}

carro.proprietario.endereco.numero = 1000

carro['proprietario']['endereco']['bairro'] = 'Av Z'

console.log(carro.calcularseguro())

## Estratégias para Criação de Objetos

// Usando a notação literal

const obj1 = {}

console.log(obj1)

//Object em JS

const obj2 = new Object

console.log(typeof Object, typeof new Object)

console.log(obj2)

// Funções Construtoras

function Produto(nome, preco, desc) {

// preco e desc estão encapsulado

this.nome = nome //Visível para fora

this.getPrecoComDesconto = () => {

return preco \* (1 - desc)

}

}

/\* Só conseguimos alterar o nome dos objetos criados

a partir de 'Produto', pois preco e desc só estão

visivéis dentro do objeto \*/

const p1 = new Produto('Caneta', 2.50, 0.15)

const p2 = new Produto('Notebook', 4999, 0.15)

console.log(p1.getPrecoComDesconto(), p2.getPrecoComDesconto())

// Função Factory

function criarFuncionario(nome, salarioBase, faltas) {

return {

nome,

salarioBase,

faltas,

getSalario() {

return (salarioBase / 30) \* (30 - faltas)

}

}

}

const f1 = criarFuncionario('Lucas', 5100, 4)

const f2 = criarFuncionario('Liana', 4000, 2)

console.log(f1.getSalario(), f2.getSalario())

// Object.create

const filha = Object.create(null)

filha.nome = 'Cecília'

console.log(filha)

// Uma função famosa que retorna Objeto...

const fromJSON = JSON.parse('{"info": "Sou um JSON"}')

console.log(fromJSON.info)

## Objetos Constantes

/\* pessoa -> 123(endereço de memória) -> {...} (Objeto)

A constante pessoa aponta para o endereço 123

que contem o objeto, o endereço é constante

mas o objeto que ele possue pode variar

\*/

const pessoa = { nome: 'Lucas' }

console.log(pessoa)

pessoa.nome = 'Liana'

console.log(pessoa)

//pessoa -> 456 -> {...}

//pessoa = { nome: 'Fulano' }

/\*Aqui, ao atribuir pessoa para outro objeto,

estamos tentando mudar o endereço, o que não pode \*/

Object.freeze(pessoa) /\* Congelamos o objeto, a partir de agora

não conseguimos alterar e nem adicionar nem deletar atributos.

Tornamos o objeto constante. \*/

pessoa.nome = 'Raul'

pessoa.end = 'Rua ABC'

delete pessoa.nome

console.log(pessoa)

//Criando objeto constante

const pessoaConstante = Object.freeze({nome: 'Lucas'})

pessoaConstante.nome = 'Liana'

console.log(pessoaConstante)

## Notação Literal

const a = 1

const b = 2

const c = 3

//Antes

const obj1 = { a: a, b: b, c: c }

//Hoje

const obj2 = { a, b, c } //Ja assume para os atributos a,b e c os valores a, b e c

console.log(obj1)

console.log(obj1)

//Criar atributos através de valores de variáveis

const nomeAttr = 'nota'

const valorAttr = 7.2

const obj3 = {}

obj3[nomeAttr] = valorAttr

console.log(obj3)

const obj4 = {[nomeAttr]: valorAttr}

console.log(obj4)

//Definição de funções dentro de objetos

//Antes

const obj5 = {

funcao1: function(){

//.....

}

}

//Hoje

const obj6 = {

funcao2(){

//.....

}

}

console.log(obj5)

console.log(obj6)

## Getters e Setters

Uma forma de se ter um encapsulamento em um objeto, é colocar os atributos privados e ter métodos que alteram e leiam os atributos de formas mais controlada.

Get: Função para acessar o valor de uma variável. Vantajoso para fazer validações/processamentos antes de devolver o valor

Set: Função para alterar o valor de uma variável

//get e set para acessar valores de forma mais controlada

const sequencia = {

\_valor: 1, // O \_ é uma convenção para sinalizar que a variável deve ser acessada apenas internamente

get valor() { return this.\_valor++ },

set valor(valor) {

//Validação

if (valor > this.\_valor) {

this.\_valor = valor

}

}

}

/\* Toda vez que tentamos acessar um valor dentro de uma função com get e set

ao acessar chama o get e ao atribuir chama o set \*/

console.log(sequencia.valor, sequencia.valor)

sequencia.valor = 1000

console.log(sequencia.valor, sequencia.valor)

sequencia.valor = 900 //Atribuindo um valor menor que o que ele tem atualmente (1000), não vai passar no set

console.log(sequencia.valor, sequencia.valor)

## Funções Importantes de Objetos

const pessoa = {

nome: 'Liana',

idade: 27,

peso: 57

}

console.log(Object.keys(pessoa)) //Retorna um array com as chaves

console.log(Object.values(pessoa))

console.log(Object.entries(pessoa)) //Sub arrays com os conjuntos chava-valor separados

Object.entries(pessoa).forEach(e => {

console.log(`${e[0]}: ${e[1]}`)

})

//Usando um destructing

Object.entries(pessoa).forEach(([chave, valor]) => {

console.log(`${chave}: ${valor}`)

})

Object.defineProperty(pessoa, 'dataNascimento', {

enumerable: true, //Se vai ser enumerada como uma chave, se false, podemos acessar mas não aparecerá como chave

writable: false, //Não permitirá mudar, semelhante ao freeze

value: '01/01/2020'

}) //Para alterar propriedades de um objeto (Ficar ou não visível)

pessoa.dataNascimento = '01/02/2020'

console.log(pessoa.dataNascimento) //Não alterou

console.log(Object.keys(pessoa))

//Object.assign (ECMAScript 2015)

const dest = { a: 1 }

const o1 = { b: 2 }

const o2 = { c: 3, a: 4 }

const obj = Object.assign(dest, o1, o2) //Pega os parâmetros de o1 e o2 e coloca em dest

console.log(obj)

Object.freeze(obj) //Não mais permite alterar parâmetros

obj.c = 123

console.log(obj)

## Herança #01

Principio da OO para reaproveitar código, onde um objeto filho herda de um objeto pai, atributos e comportamentos.

/\* Sempre que puder, opte pela composição ao invez da herança

Em JavaScrit a herança é baseada em protótipo.

Um objeto pai (protótipo) tem seus atributos e comportamentos

assim como o objeto filho; ao tentar acessar um atributo do

objeto filho, e não existir, a busca é direcionada para o ob-

jeto pai, não achou, vai para o objeto avô, e assim por diante

[[prototype]] != prototype

Conceito de protótipo != Propriedade prototype (Que existem dentro das funções)

\*/

const ferrari = {

modelo: 'F40',

velMax: 324

}

const volvo = {

modelo: 'V40',

velMax: 200

}

console.log(ferrari.\_\_proto\_\_) /\* .\_\_proto\_\_ tenta acessar um objeto,

se não encontrar tentar acessar no seu protótipo (objeto pai) \*/

console.log(ferrari.\_\_proto\_\_ === Object.prototype) // .prototype está presenta apenas dentro de funções

console.log(volvo.\_\_proto\_\_ === Object.prototype)

console.log(Object.prototype.\_\_proto\_\_ === null)

function MeuObjeto(){}

console.log(typeof Object, typeof MeuObjeto)

console.log(Object.prototype, MeuObjeto.prototype)

## Herança #02

// Cadeia de protótipos

Object.prototype.attr0 = '0' //Não faça isso em casa

const avo = { attr1: 'A' } //Aponta para Object.prototype

const pai = { \_\_proto\_\_: avo, attr2: 'B', attr3: 'C1' }

const filho = { \_\_proto\_\_: pai, attr3: 'C' }

// filho tem o pai como protótipo, pai tem o avo como protótipo e avo tem o Object.prototype

console.log(filho.attr0, filho.attr1, filho.attr3)

/\* Se um objeto filho tem um atributo igual ao atributo de um objeto

pai, o atributo do filho, caso tenha valor diferente, sobrescreve o

do pai \*/

const carro = {

velAtual: 0,

velMax: 200,

acelerar(delta) {

if (this.velAtual + delta <= this.velMax) {

this.velAtual += delta

} else {

this.velAtual = this.velMax

}

},

status() {

return `${this.velAtual} Km/h de ${this.velMax} Km/h`

}

}

const ferrari = {

modelo: 'F40',

velMax: 324 /\* shadowing - Sombreamento do parâmetro

existente no objeto pai (carro). \*/

}

const volvo = {

modelo: 'V40',

status() {

return `${this.modelo}: ${super.status()}`

/\*O super, vai trazer o status de um objeto

pai \*/

}

}

//Estabelecendo a relação de protótipos

Object.setPrototypeOf(ferrari, carro) //ferrari tem carro como protótipo

Object.setPrototypeOf(volvo, carro)

/\* Ao chamar a string do objeto, vai retornar só os parâmetros

e atributos definidos dentro desse objeto, os parâmetros e

atributos dos objetos pai, não. \*/

console.log(ferrari)

console.log(volvo)

volvo.acelerar(100)

console.log(volvo.status())

ferrari.acelerar(300)

console.log(ferrari.status())

volvo.acelerar(150)

console.log(volvo.status())

## Herança #03

const pai = { nome: 'Dedé', corCabelo: 'castanho' }

const lucas = Object.create(pai) //Cria o objeto lucas tendo pai como parâmetro

lucas.nome = 'Lucas'

console.log(lucas)

const vinicius = Object.create(pai, {

nome: { value: 'Vinicius', writable: false, enumerable: true }

})

console.log(vinicius.nome)

vinicius.nome = 'Xarope'

console.log(`${vinicius.nome} tem cabelo ${vinicius.corCabelo}`)

console.log(Object.keys(lucas)) //Não lista os parâmetros herdados

console.log(Object.keys(vinicius))

//Podemos listar os parÂmetros herdados com um for in

for (let key in lucas) {

vinicius.hasOwnProperty(key) ?

console.log(`Própria: ${key}`) : console.log(`Por herança: ${key}`)

}

## Herança #04

function MeuObjeto() {}

console.log(MeuObjeto.prototype)

const obj1 = new MeuObjeto

const obj2 = new MeuObjeto

/\* obj1 e obj2 vão apontar para object.prototype ou para MeuObjeto.prototype ? \*/

console.log(obj1.\_\_proto\_\_ === obj2.\_\_proto\_\_)

/\* Todos os objetos criados a partir da mesma função construtora

apontam para o mesmo protótipo ? SIM\*/

console.log(obj1.\_\_proto\_\_ === MeuObjeto.prototype)

/\* Quando criamos um objeto a partir de uma função construtora

usando 'new', o protótipo que criamos aponta para a o prototype da

função construtora \*/

MeuObjeto.prototype.nome = 'Liana'

MeuObjeto.prototype.falar = function (){

console.log(`Bom dia! Meu nome é ${this.nome}!`)

}

obj1.falar()

obj2.nome = 'Lucas'

obj2.falar()

const obj3 = {}

obj3.\_\_proto\_\_ = MeuObjeto.prototype /\* Mudando a referência do obj3

Aqui, não criamos obj3 a partir de uma função construtora, logo a

referência não é 'MeuObjeto.prototype' \*/

obj3.nome = 'Obj3'

obj3.falar()

//Resumo

console.log((new MeuObjeto).\_\_proto\_\_ === MeuObjeto.prototype)

console.log(MeuObjeto.\_\_proto\_\_ === Function.prototype)

/\* A função tambem tem uma referência para \_\_protp\_\_, que aponta para

Function.prototype \*/

console.log(Function.prototype.\_\_proto\_\_ === Object.prototype)

console.log(Object.prototype.\_\_proto\_\_ === null)

## Herança #05

console.log(typeof String)

console.log(typeof Array)

console.log(typeof Object)

String.prototype.reverse = function () {

return this.split('').reverse().join('')

/\* Acessamos os parâmetros dentro do protótipo com

o 'this' \*/

}

console.log('Escola Cod3r'.reverse())

//O método reverse não tem no protótipo, nós adicionamos com a função

Array.prototype.first = function () {

return this[0]

}

console.log([1, 2, 3, 4, 5].first())

console.log(['a', 'b', 'c', 'd', 'e'].first())

//Podemos substituir metodos que ja existem

String.prototype.toString = function () {

return 'Lascou Tudo'

}

console.log('Escola Cod3r'.reverse()) /\* Não vai mais receber 'Escola Cod3r' como string, pois alteramos o método 'toString', ou seja, nossa string agora é

'Lascou Tudo' \*/

## Herança #06

function Aula(nome, videoID) {

this.nome = nome

this.videoID = videoID

}

const aula1 = new Aula('Bem Vindo', 1)

const aula2 = new Aula('Até Breve', 254)

console.log(aula1, aula2)

/\* Quando temos uma função construtora (Aula) e usamos o

operador 'new', o protótipo desse objeto aponta para Aula.prototype \*/

// Simulando o new - O que é feito internamente

function novo(f, ...params) { // ... = SpreadRest

const obj = {}

obj.\_\_proto\_\_ = f.prototype

f.apply(obj, params) //

return obj

}

const aula3 = novo(Aula, 'Bem Vindo', 1)

const aula4 = novo(Aula, 'Até Breve', 254)

console.log(aula3, aula4)

## Evitando Modificações

Por ser fracamente tipada, JavaScript permite que atributos de objetos sofram modificações, e essas modificações as vezes são indesejáveis, e podemos contornar, para ter maior previsibilidades de determinados objetos.

## JSON vs Objeto

// JSON - JavaScript Object Notation

/\* Formato de dados mais usados para interoperabilidade entre sistemas \*/

const obj = { a: 1, b: 2, c: 3, soma(a, b, c) { return a + b + c } }

//Transformar um Objeto em JSON

console.log(JSON.stringify(obj))

//Transformar um JSON em Objeto

console.log(JSON.parse('{"a":1, "b":2, "c":2}'))

/\* Os atrubutos para ter um JSON válido tem que ter

os mesmos em aspas dupla("") \*/

console.log(JSON.parse('{"a":1, "b": "string", "c": true, "d": {}, "e": []}'))

## Classe #01

Por traz dos panos, as classes são funções construtivas, temos uma outra forma de criar funções construtoras

class Lancamento {

//Nossa função construtora

constructor(nome = 'Genérico', valor = 0) {

this.nome = nome

this.valor = valor

}

}

class CicloFinanceiro {

constructor(mes, ano) {

this.mes = mes

this.ano = ano

this.lancamentos = []

}

addLancamentos(...lancamentos) {

lancamentos.forEach(l => this.lancamentos.push(l))

}

sumario() {

let valorConsolidado = 0

this.lancamentos.forEach(l => {

valorConsolidado += l.valor

})

return valorConsolidado

}

}

const salario = new Lancamento('Salário', 45000)

const contaluz = new Lancamento('Luz', -220)

const contaagua = new Lancamento('Luz', -70)

const contas = new CicloFinanceiro(2, 2021)

contas.addLancamentos(salario, contaluz, contaagua)

console.log(contas.sumario())

## Classe #02

//Herança com classes JavaScript

class Avo {

constructor(sobrenome) {

this.sobrenome = sobrenome

}

}

//Criando uma classe que herda outra

class Pai extends Avo {

constructor(sobrenome, profissao = 'Analista de Sistemas') {

super(sobrenome) //Chamada do construtor da classe Avo

this.profissao = profissao

}

}

class Filho extends Pai {

constructor() {

super('Peixoto')

}

}

const filho = new Filho

console.log(filho)

# Array

## Visão Geral

Em JavaScript, o array é um objeto, um objeto com características próprias, que se organiza em uma estrutura indexada, com índices, sem valores, começando do ‘0’.

// Arrays em JS, são heterogenos, suportando tipos diferentes de dados

console.log(typeof Array, typeof new Array, typeof [])

let aprovados = new Array('Lucas', 'Liana', 'Fulano', 'Ciclano') //Forma não recomendada

console.log(aprovados)

aprovados = ['Lucas', 'Liana', 'Fulano', 'Ciclano']

console.log(aprovados[0])

console.log(aprovados[1])

console.log(aprovados[2])

console.log(aprovados[3])

console.log(aprovados[4]) //undefined

aprovados[4] = 'Beltrano'

console.log(aprovados)

aprovados.push('Zézinho')

console.log(aprovados.length)

aprovados[10] = 'Luluzinha' //Vai criar o elemento de posição 9, e todos entre 6 e 8, serão undefined

console.log(aprovados.length)

console.log(aprovados)

aprovados.sort() //Ordenação em ordem alfabética ou sequencial - Altera o array original

console.log(aprovados)

delete aprovados[1]

console.log(aprovados) // Coloca undefined na posição deletada

aprovados = ['Lucas', 'Liana', 'Fulano', 'Ciclano']

aprovados.splice(1, 2) //Serve para adicionar, remover e adicionar e remover ao mesmo tempo

/\* (1, 1) A partir do element0 1, exclui 1

(1, 2) A partir do elemento 1, exclui 2

\*/

console.log(aprovados)

aprovados = ['Lucas', 'Liana', 'Fulano', 'Ciclano']

aprovados.splice(1, 2, 'Luluzinha', 'Zézinho')

/\* (1, 2, 'Luluzinha', 'Zézinho') A partir do elemento 1,

exclui 2 e adiciona 'Luluzinha' e 'Zézinho' \*/

console.log(aprovados)

aprovados = ['Lucas', 'Liana', 'Fulano', 'Ciclano']

aprovados.splice(1, 0, 'Luluzinha', 'Zézinho')

/\* (1, 2, 'Luluzinha', 'Zézinho') A partir do elemento 1,

exclui 0 (nenhuma exclusão) e adiciona 'Luluzinha' e 'Zézinho',

a partir da posição 1 \*/

console.log(aprovados)

## Array: Métodos Importantes

const pilotos = ['Vettel', 'Alonso', 'Raikkonen', 'Massa']

console.log(pilotos)

/\* A constante pilotos sempre vai apontar para o array definido, mas

o array pode ser alterado (Incluir/Excluir/Modificar itens) \*/

pilotos.pop() //Remove o último elemento do array

console.log(pilotos)

pilotos.push('Verstappen') //Inclui o elemento passado

console.log(pilotos)

pilotos.shift() //Remove o 1º Elemento da lista

console.log(pilotos)

pilotos.unshift('Hamilton') //Adiciona um elemento na 1ª Posição do array

console.log(pilotos)

pilotos.splice(2, 0, 'Bottas', 'Massa') // Adiciona na posição 2, o 'Bottas' e o 'Massa', remove 0

console.log(pilotos)

pilotos.splice(3, 1) //O Elemento na 3ª posição é removido, remove 1

console.log(pilotos)

const algunsPilotos1 = pilotos.slice(2) //Cria um novo array a partir do indice 2 de pilotos

console.log(algunsPilotos1)

const algunsPilotos2 = pilotos.slice(1, 4) //Cria um novo array, pegando do elemento de indice 1 até indice 3 (4 não entra)

console.log(algunsPilotos2)

## Simulando Array com Objeto

const quaseArray = { 0: 'Liana', 1: 'Lucas', 2: 'Cecília', 3: 'César' }

console.log(quaseArray)

Object.defineProperty(quaseArray, 'toString', {

value: function () { return Object.values(this) },

enumerable: false

})

console.log(quaseArray[0])

const meuArray = ['Liana', 'Lucas', 'Cecília', 'César']

console.log(meuArray)

console.log(quaseArray.toString())

## Foreach #01

const aprovados = ['Fulano', 'Ciclano', 'Beltrano', 'Ramiro', 'Pazuelo']

aprovados.forEach(function (nome, indice) {

console.log(`${indice + 1}) ${nome}`)

})

/\* O foreach vai percorrer o array, um laço, onde em cada

iteração, conseguimos acessar o item do array.

O primeiro parâmetro é o nome, o segundo o índice e o terceiro

o próprio array.

\*/

//Passando apenas o nome com uma arrow function

aprovados.forEach(nome => console.log(nome))

//Passando uma função pré-definida

const exibirAprovados = aprovado => console.log(aprovado)

aprovados.forEach(exibirAprovados)

## Foreach #02

//Desafio - Implementar um método igual o forEach (forEach2)

Array.prototype.forEach2 = function (callback) { //callback is the function

//Scrolling through the indices

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

//Calling the functions

callback(this[i], i, this)

}

}

const aprovados = ['Fulano', 'Ciclano', 'Beltrano', 'Ramiro', 'Pazuelo']

aprovados.forEach2(function (nome, indice) {

console.log(`${indice + 1}) ${nome}`)

})

## Map #01

/\*Dentro do método map existe um laço for, e a ideia

é fazer uma transformação no array, mapear o array para

outro array do mesmo tamanho com os dados transformados \*/

const nums = [1, 2, 3, 4, 5]

//map = for com propósito

let resultado = nums.map(function (e) {

return e \*\* 2 //e^2

})

console.log(resultado)

const soma10 = e => e + 10

const triplo = e => e \* 3

const paradinheiro = e => `R$ ${parseFloat(e).toFixed(2).replace('.', ',')}`

resultado = nums.map(soma10).map(triplo).map(paradinheiro)

console.log(resultado)

## Map #02

const carrinho = [

'{ "nome": "Borracha", "preco": 3.45 }',

'{ "nome": "Caderno", "preco": 13.90 }',

'{ "nome": "Kit Lápis", "preco": 41.22 }',

'{ "nome": "Caneta", "preco": 7.50 }',

]

//Desafio - Retornar um array apenas com os preços usando map

const precos = carrinho.map(function(e){

let obj = JSON.parse(e)

let preco = obj.preco

return preco

})

console.log(precos)

## Map #03

Array.prototype.map2 = function (callback) {

const newArray = []

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

newArray.push(callback(this[i], i , this))

}

return newArray

}

const carrinho = [

'{ "nome": "Borracha", "preco": 3.45 }',

'{ "nome": "Caderno", "preco": 13.90 }',

'{ "nome": "Kit Lápis", "preco": 41.22 }',

'{ "nome": "Caneta", "preco": 7.50 }',

]

//Desafio - Implementar um método (map2) semelhante ao map

const precos = carrinho.map2(function (e) {

let obj = JSON.parse(e)

let preco = obj.preco

return preco

}

)

console.log(precos)

## Filter #01

/\*O filter é para filtrar um array e obter, por exemplo, um sub-array

O array retornardo vai ser menor ou igual ao array original

\*/

const produtos = [

{nome: 'Notebook', preco: 4900, fragil: true},

{nome: 'Desktop', preco: 6999, fragil: false},

{nome: 'Ipad', preco: 2199, fragil: true},

{nome: 'Copo de Vidro', preco: 12.49, fragil: true},

{nome: 'Copo de Plástico', preco: 18.99, fragil: false}

]

/\* o filter, em cada iteração no array, vai receber um retorno

true ou false, se for true, entra no array novo

\*/

//Filtrando apenas produtos frageis e aciam de 500 reais

const frageis = produtos.filter(function(p){

if (p.preco > 500 && p.fragil === true) {

return true

} else {

return false

}

})

console.log(frageis)

## Filter #02

// Desafio - Implementar o próprio metodo fitler

Array.prototype.filter2 = function (callback) {

const newArray = []

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

let callbackreturn = callback(this[i], i, this)

if (callbackreturn == true) {

newArray.push(this[i])

} else {

continue

}

}

return newArray

}

const produtos = [

{ nome: 'Notebook', preco: 4900, fragil: true },

{ nome: 'Desktop', preco: 6999, fragil: false },

{ nome: 'Ipad', preco: 2199, fragil: true },

{ nome: 'Copo de Vidro', preco: 12.49, fragil: true },

{ nome: 'Copo de Plástico', preco: 18.99, fragil: false }

]

/\* o filter, em cada iteração no array, vai receber um retorno

true ou false, se for true, entra no array novo

\*/

//Filtrando apenas produtos frageis e aciam de 500 reais

const frageis = produtos.filter2(function (p) {

if (p.preco > 500 && p.fragil === true) {

return true

} else {

return false

}

})

console.log(frageis)

## Reduce #01

/\* Função que serve para transformar um array em um array, em um element

número, etx... Temos um parâmetro que vai sendo acumulado de chamada

em chamada, ou seja, o resultado de uma função reduce é sempre passado

para a próxima iteração. O primeiro reduce recebe os dois primeiros

elementos (0 e 1), transforma e o resultado é passado para a próxima

e assim por diante. O elemento acumulador é o ValorInicial, que se não

for chamado, vai ser o elemento de indice 0, depois o de indice 1 e assim

por diante.

\*/

const alunos = [

{ nome: 'João', nota: 7.3, bolsista: false },

{ nome: 'Maria', nota: 9.2, bolsista: true },

{ nome: 'Pedro', nota: 9.8, bolsista: false },

{ nome: 'Ana', nota: 8.7, bolsista: true }

]

const resultado = alunos.map(a => a.nota).reduce(function (acumulador, atual) {

console.log(acumulador, atual)

return acumulador + atual

}, 0) // 0 = acumulador passado

console.log(resultado)

## Reduce #02

const alunos = [

{ nome: 'João', nota: 7.3, bolsista: false },

{ nome: 'Maria', nota: 9.2, bolsista: false },

{ nome: 'Pedro', nota: 9.8, bolsista: false },

{ nome: 'Ana', nota: 8.7, bolsista: false },

{ nome: 'Cecília', nota: 10, bolsista: false },

{ nome: 'César', nota: 10, bolsista: false },

{ nome: 'Josefina', nota: 6.8, bolsista: true },

{ nome: 'Arlindo', nota: 3.8, bolsista: false }

]

//Desafio 1 - Com o reduce, retornar se todos os alunos são bolsistas

const bolsistas = alunos.map(aluno => aluno.bolsista)

console.log(bolsistas.reduce(function (acumulador, atual) {

let check = acumulador && atual

return check

}

)

)

// Com arrow function - console.log(bolsistas.reduce((acumulador, atual) => acumulador && atual))

//Desafio 2 - Com o reduce, retornar se algum aluno é bolsista

console.log(bolsistas.reduce(function (acumulador, atual) {

let check = acumulador || atual

return check

}))

## Reduce #03

Array.prototype.reduce2 = function(callback, ValorInicial) {

const indiceInicial = ValorInicial ? 0 : 1

let acumulador = ValorInicial || this[0] //ValorInicial o 1º elemento

for (let i = indiceInicial; i < this.length; i++){

acumulador = callback(acumulador, this[i], i, this)

}

return acumulador

}

const alunos = [

{ nome: 'João', nota: 7.3, bolsista: false },

{ nome: 'Maria', nota: 9.2, bolsista: true },

{ nome: 'Pedro', nota: 9.8, bolsista: false },

{ nome: 'Ana', nota: 8.7, bolsista: true }

]

const resultado = alunos.map(a => a.nota).reduce2(function (acumulador, atual) {

console.log(acumulador, atual)

return acumulador + atual

}, 0) // 0 = acumulador passado

console.log(resultado)

## Declarativo x Imperativo

// Código Imperativo x Declarativo

const alunos = [

{ nome: 'João', nota: 7.3 },

{ nome: 'Maria', nota: 9.2 },

{ nome: 'Pedro', nota: 9.8 },

{ nome: 'Ana', nota: 8.7 },

{ nome: 'Cecília', nota: 10 },

{ nome: 'César', nota: 10 },

{ nome: 'Josefina', nota: 6.8 },

{ nome: 'Arlindo', nota: 3.8 }

]

//Calculando a média das notas - Minha forma

const notas = alunos.map(a => a.nota)

let total = 0

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

total += notas[i]

}

const media = total/notas.length

console.log(`Média Minha ${media}`)

//Calculando a médias das notas - Forma Imperativa

let total1 = 0

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

total1 += alunos[i].nota

}

const media1 = total1/alunos.length

console.log(`Média Imperativo ${media1}`)

//Calculando a média das notas - Forma Declarativa

const getNota = aluno => aluno.nota

const soma = (total, atual) => total + atual

const total2 = alunos.map(getNota).reduce(soma)

const media2 = total2/alunos.length

console.log(`Média Declarativo ${media2}`)

/\* A abordagem declarativa é mais interessante para aproveitamento

de código e redução de código \*/

## Concat

const filhos = ['Cesar', 'Rubens']

const filhas = ['Cecília', 'Alice']

const todos = filhas.concat(filhos, 'Fulano')

console.log(todos)

console.log([].concat([1, 2], [3, 4], 5, [[6, 7]]))

## FlatMap

// Flatmap = map associado com concat

const escola = [{

nome: 'Turma M1',

alunos: [{

nome: 'Gustavo',

nota: 7.1

}, {

nome: 'Cecília',

nota: 10

}]

}, {

nome: 'Turma M2',

alunos: [{

nome: 'Cesar',

nota: 9.9

}, {

nome: 'Verônica',

nota: 6.8

}]

}]

//Extrair todos as notas

const getNotaDoAluno = aluno => aluno.nota

const getNotasDaTurma = turma => turma.alunos.map(getNotaDoAluno)

const notas1 = escola.map(getNotasDaTurma)

console.log(notas1) //Não obtem todas as notas em um único array

//Usando flatMap

Array.prototype.flatMap = function(callback) {

return Array.prototype.concat.apply([], this.map(callback))

}

const notas2 = escola.flatMap(getNotasDaTurma)

console.log(notas2)

# Node

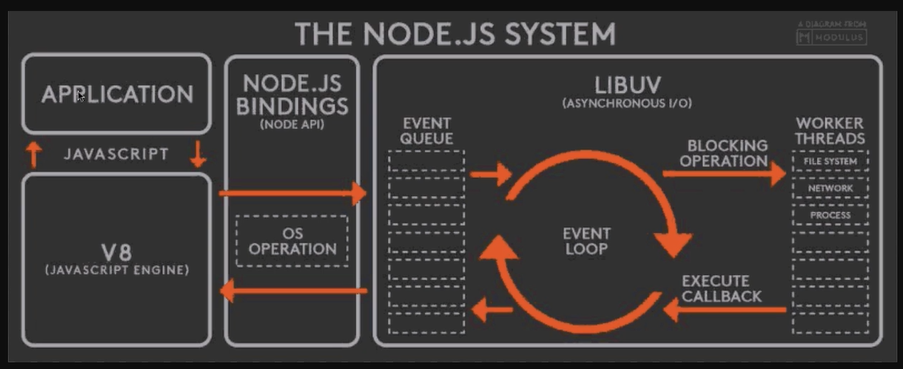
## Visão Geral

Cliente: Browser Executando

Servidor: Que prove as API’s, acesso a banco de dados

JavaScript é a linguagem que dá a possibilidade de rodar tanto no browser quanto no *server-side*, servidor (backend).

JavaScript, inicialmente nasceu para trabalhar com ambientes assíncronos, pois, ao estar no browser, sempre está se demandando coisas do servidor remoto



O Node é um runtine JavaScript composto pelo V8 e o Libuv

V8: Interpretador JavaScript do chrome, que vai ler o código e vai fazer chamadas usando a libuv, que faz toda a parte do assincronismo (io assíncrono, sempre que precisamos acessar arquivos, banco de dados, tudo envolvendo entradas e saídas, e um tempo de espera).

## Sistema de Módulos

Hoje as aplicações são organizadas em arquivos, diferente de antigamente, onde tudo ficava em um único arquivo. No final, tudo é compactado em um único arquivo para enviar ao browser.

Isso é feita para otimizar a comunicação dos arquivos com o browser.

No backend, será também organizado em pastas, mas não vai ser necessário compactar em um arquivo para enviar ao browser, o que é enviado ao browser, é um arquivo JSON.

Em node, um arquivo representa um módulo, que tem uma certa interioridade, não deixa o que está escrito lá, visível fora do modulo, para isso, será necessário exportar para fora do módulo.

Na versão mais atual do node, existem dois padrões de módulo, o commonsJs e o ImportExport (ECMAScript 2015).

## Usando Módulos de Terceiros

Podemos usar módulos de terceiros, importando-os nos nossos módulos

Por padrão, ao instalarmos módulos, eles vão para a pasta node\_modules, que devemos baixar (npm install lodash).

Para instalar uma lib fora do node/modules, precisamos informar que ela é global(g) npm i -g nodemon

O nodemon consegue executar módulos e interpretar as atualizações em tempo real (durante a execução).

## Sistema de Módulos: Require

Ao criar uma pasta dentro de node\_modules, por padrão, será procurado nessa pasta, o arquivo index.js, e aqui, podemos deixar algo disponível para quando, alguém procurar o módulo.

Além dos módulos instalados (em node\_modules), podemos importar os módulos que já vem instalados no node (módulos core).

Como o require, ao chamar um módulo, vai procurar dentro de index.js, podemos colocar em index.js tudo que queremos deixar disponível, ou seja, ser uma forma única de acessar outros arquivos.

## Sistema de Módulos: Exports

// Como exportar variáveis ? Deixa-las fora do arquivo

console.log(module.exports === this)

console.log(module.exports === exports)

//Visível fora do arquivo:

this.a = 1

module.exports.b = 2

exports.c = 3

exports = null //Não vai alterar o retorno, quem é exportado para fora é o module.exports

console.log(module.exports) //Se mantem o mesmo

exports = {

nome: 'Liana',

status: 'Linda'

}

console.log(module.exports)

/\* Mudamos a referência de exports duas vezes (linha 11 e 15),

mas ele continua apontando para o mesmo objeto. Para ele apontar

para outro objetos, precisamos usar o module.exports.

O this e o exports são 2 referentes para o objeto que module.exports aponta

\*/

## Arquivo package.json

Arquivo padrão de projetos com Node.js, que tem o objetivo de descrever o seu projeto e listar as dependências. Para criar esse arquivo, precisamos executar o ‘npm init’, e informar alguns parâmetros desse projeto, npm init -y responde sim para todas as perguntas.

npm i --save axios: Instala o node\_modules com a lib axios e o (--save) cria um espaço em package.json para listar as dependências.

npm install --save-dev [axios@0.17.11](mailto:axios@0.17.11) -E: dev é para instalar a dependência apenas em desenvolvimento, não sobe para produção.

O axios é um cliente http, ele faz requisições para obter informações de algo que está remoto.

O package-lock.json é criado para garantir que os pacotes que vão subir são os instalados.

## Instância Única vs Nova Instância

Instância: Objeto criado a partir de uma função construtora.

Cache: Área de memória onde é mantida uma cópia temporária de dados armazenados em um meio de acesso mais lento, com o objetivo de acelerar a recuperação dos dados.

Como o node faz cache de modos que são importados?

Toda vez que importamos um módulo no node, por padrão, esse modulo é cacheado. Vamos usar um recurso que vimos no capítulo de funções, que é gerar um objeto toda vez que um modulo for importado, para isso usaremos a função factory.

**instânciaUnica.js:**

// node faz cache - mantem essas instâncias como únicas

module.exports = {

valor: 1,

inc() {

this.valor++

}

}

**instânciaNova.js:**

/\* Como retornar uma nova instância, ja que,

por padrão, o node retorna um cache?

Podemos usar uma factory, já que retorna uma nova

instância (objeto)

\*/

module.exports = () => {

return {

valor: 1,

inc() {

this.valor++

}

}

}

**instânciaCliente.js:**

const contadorA = require('./instanciaUnica') /\*Aqui, por ser instância única, o node faz cache ao

importarmos com require, ou seja, se o objeto ja tiver sido criado, ele retorna a mesma instância \*/

const contadorB = require('./instanciaUnica')

//const contadorC = require('./instanciaNova') //Retorna uma instância nova (Uma função)

const contadorC = require('./instanciaNova')() //Retorna um objeto

const contadorD = require('./instanciaNova')()

contadorA.inc()

contadorA.inc()

console.log(contadorA.valor, contadorB.valor)

/\* Como é criado um cache ao importar o modula 'instanciaUnica',

tudo que ocorre em A, B sente, e vice-versa. \*/

console.log('\n')

contadorC.inc()

contadorC.inc()

console.log(contadorC.valor, contadorD.valor)

/\*Aqui, são novas instâncias que foram criadas a partir de uma factory,

logo, o que ocorre em C, não é sentido por D \*/

/\*

No caso de instância única, os contadores A e B estariam apontando para o mesmo

endereço de memória (gerado ao importar o próprio objeto atribuído a module.exports) e,

consequentemente, uma alteração em qualquer um deles estaria sendo feita diretamente no

objeto que está guardado naquele endereço de memória. Desse modo, invocar a função por meio

de A ou B ou imprimir algum de seus valores é o mesmo que fazê-lo a partir do próprio objeto importado.

Já no caso dos contadores C e D, importa-se uma função cujo retorno é um objeto e invoca-se ela já dentro

do módulo do cliente. Os objetos guardados a contadorC e contadorD estariam, então, sendo criados no

momento em que a função é invocada. É como se cada contador estivesse recebendo uma declaração literal de

um novo objeto, igual ao definido na função. Com isso, define-se um endereço de memória para cada contador.

Não é o próprio objeto alocado em module.exports do outro módulo que está sendo importado,

mas uma função que retorna um objeto.

\*/

## Objeto Global do node

O padrão, é utilizarmos o sistema de módulos, mas podemos utilizar do objeto global em alguns casos.

**Em global.js:**

/\*console.log(global)

Object [global] {

global: [Circular \*1],

clearInterval: [Function: clearInterval],

clearTimeout: [Function: clearTimeout],

setInterval: [Function: setInterval],

setTimeout: [Function: setTimeout] {

[Symbol(nodejs.util.promisify.custom)]: [Function (anonymous)]

},

queueMicrotask: [Function: queueMicrotask],

clearImmediate: [Function: clearImmediate],

setImmediate: [Function: setImmediate] {

[Symbol(nodejs.util.promisify.custom)]: [Function (anonymous)]

}

}

\*/

//Criando um objeto dentro de global

global.MinhaApp = {

saudacao() {

return 'Estou visível em todos módulos'

},

nome: 'Sistema Legal'

}

**Em globalCliente.js:**

require('./global') //Carregando o objeto global para todos os outros módulos

console.log(MinhaApp.saudacao())

MinhaApp.nome = 'Eita, mudou'

console.log(MinhaApp.nome)

## Entendendo o this

O this aponta para module.export, e podemos exportar usando ele

console.log(this === global)

console.log(this === module)

console.log(this === module.exports)

console.log(this === exports)

//Acessando o this dentro de uma função que está no módulo

function logThis(){

console.log('Dentro de uma função...')

console.log(this === exports)

console.log(this === module.exports)

console.log(this === global)

}

logThis() /\*Dentro de uma função, o this não aponta para

exports nem module.exports, mas fora aponta.

Dentro da função, this aponta para global

\*/

## Passando parâmetros entre os módulos

**Em passandoParametros.js:**

module.exports = function (...nomes) {

return nomes.map(nome => `Boa semana ${nome}!`)

}

**Em passandoParametrosCliente.js:**

const saudacoes = require('./passandoParametros')('Liana', 'Lucas', 'Cecília', 'César')

console.log(saudacoes)

## Instalando Deps & Scripts

Em cursojs/node/funcionários/package.json:

{

"name": "funcionarios",

"version": "1.0.0",

"description": "Aula package.json",

"main": "funcionarios.js",

"scripts": {

"start": "nodemon",

"dev": "nodemon",

"test": "echo \"Error: no test specified\" && exit 1"

},

"author": "Lucas Peixoto",

"license": "ISC",

"dependencies": {

"axios": "^0.21.1"

}

}

Podemos executar os Scripts padrão no terminal:

npm start, npm dev, etc…

Podemos tambem executar Scripts próprios:

npm run teste\_api

## Lendo Arquivos

const fs = require('fs') //fs = file system

const caminho = \_\_dirname + '/arquivo.json'

/\* \_\_dirname, é uma variável que está presente em todos

os módulos, que representa o diretório atual \*/

//sincrono... Tem que ler completamente o arquivo, para só aí liberar o EventLoop

const conteudo = fs.readFileSync(caminho, 'utf-8')

console.log(conteudo)

//assincrono... Chama uma callback quando o arquivo for carregado

fs.readFile(caminho, 'utf-8', (err, conteudo) => {

const config = JSON.parse(conteudo) //JSON

console.log(`${config.db.user}:${config.db.password}`)

})

//Outra forma de ler JSON

const config = require('./arquivo.json')

console.log(config.db)

//Ler conteúdo de um diretório

fs.readdir(\_\_dirname, (err, arquivos) =>{

console.log('Conteúdo da pasta...')

console.log(arquivos)

})

## Escrevendo Arquivos

const fs = require('fs')

const produto = {

nome: 'Celular',

preco: 1249.99,

desconto: 0.15

}

fs.writeFile(\_\_dirname + '/arquivoGerado.json', JSON.stringify(produto), err =>

{

console.log(err || 'Arquivo salvo!')

})

## Frameworks Web

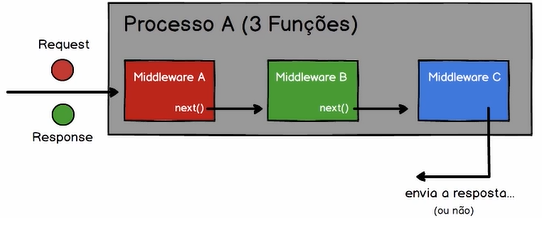
Os frameworks web dão uma estrutura para construção das nossas aplicações web.

O Express é um framework, que vai ajudar a construir os *web services* no backend.

## Padrão Middleware #01

Tem como ideia central separar processos em pequenos passos, sem amarração entre os passos, ou seja, não tem um acoplamento de um passo em outro, ou seja, um passo não chama o outro, para isso teremos uma função para isso, uma função next(), que conhece o próximo passo da cadeia.

Figura 1. Estruturação básica do padrão de projeto middleware.



## Padrão Middleware #02

//middleware pattern (chain of responsability)

const passo1 = (ctx, next) => {

ctx.valor1 = 'mid1'

next()

}

const passo2 = (ctx, next) => {

ctx.valor2 = 'mid2'

next()

}

const passo3 = ctx => ctx.valor3 = 'mid3'

const exec = (ctx, ...middlewares) => {

const execPasso = indice => {

middlewares && indice < middlewares.length &&

middlewares[indice](ctx, () => execPasso(indice + 1))

}

execPasso(0)

}

const ctx = {}

exec(ctx, passo1, passo2, passo3)

console.log(ctx)

## Instalando o Postman

Princípio de API’s Rest: Dar um JSON como resposta

O postman, vai permitir fazer requisições de forma mais flexível, permitindo testar as diferentes formas de se fazer requisição (get, post, etc...).

## Projeto: API com Express

**Em package.json:**

{

"name": "projeto",

"version": "1.0.0",

"description": "",

"main": "src/servidor.js",

"scripts": {

"start": "nodemon",

"test": "echo \"Error: no test specified\" && exit 1"

},

"keywords": [],

"author": "",

"license": "ISC",

"dependencies": {

"express": "4.16.2"

},

"devDependencies": {

"nodemon": "1.14.11"

}

}

**Em sevidor.js:**

const porta = 3003 /\*Porta é um processo dentro do computador

A porta é necessária sempre que executamos processos que precisam

se comunicar com outro computador. A porta é a forma de selecionar

qual processo vai atender determinada requisição. É a associação de

um número que comunica um processo pela rede

\*/

const express = require('express')

const app = express()

const bodyParser = require('body-parser')

const bancoDeDados = require('./bancoDeDados')

app.use(bodyParser.urlencoded({ extended: true }))

app.get('/produtos', (req, res, next) => {

res.send(bancoDeDados.getProdutos()) //converte para JSON (metodo sendo faz isso)

})

app.get('/produtos/:id', (req, res, next) => {

res.send(bancoDeDados.getProduto(req.params.id))

})

app.post('/produtos', (req, res, next) => {

const produto = bancoDeDados.salvarProduto({

nome: req.body.nome,

preco: req.body.preco

})

res.send(produto) //Retorna um JSON

})

app.put('/produtos/:id', (req, res, next) => {

const produto = bancoDeDados.salvarProduto({

id: req.params.id,

nome: req.body.nome,

preco: req.body.preco

})

res.send(produto) //Retorna um JSON

})

app.delete('/produtos/:id', (req, res, next) => {

const produto = bancoDeDados.excluirProduto(req.params.id)

res.send(produto) //Retorna um JSON

})

app.listen(porta, () => {

console.log(`Servidor está executando na porta ${porta}`)

})

npm i --save body-parser: Vai fazer um ‘parser’ no body da requisição e vai entregar os dados corretos no lado do servidor. No lado do servidor, precisamos transformar em objeto para acessar.

**Em bancoDeDados.js:**

const sequence = {

\_id: 1,

get id() { return this.\_id++ }

}

const produtos = {}

function salvarProduto(produto) {

if (!produto.id) { produto.id = sequence.id }

produtos[produto.id] = produto

return produto

}

function getProduto(id) {

return produtos[id] || {}

}

function getProdutos() {

return Object.values(produtos)

}

function excluirProduto(id) {

const produto = produtos[id]

delete produtos[id]

return produto

}

module.exports = { salvarProduto, getProduto, getProdutos, excluirProduto }

## Tarefas Agendadas com temporizador

const schedule = require('node-schedule')

const tarefa1 = schedule.scheduleJob('\*/5 \* 8 \* \* 3', function () {

console.log('Executando Tarefa 1!', new Date().getSeconds())

})

// \*/5 \* 8 \* \* 3 - De 5 em 5 segundos, as 8hs, ignorando dia e mes, na quarta feira (0-domingo, 1-segunda, etc...)

setTimeout(function () {

tarefa1.cancel()

console.log('Cancelando a Tarefa 1')

}, 20000)

const regra = new schedule.RecurrenceRule()

regra.dayOfWeek = [new schedule.Range(1, 5)]

regra.hour = 8

regra.minute = 5

regra.second = 10

const tarefa2 = schedule.scheduleJob(regra, function () {

console.log('Executando Tarefa 2!', new Date().getSeconds())

})

## Process: Entrada e Saída Padrão

/\* Obtendo dados do teclado \*/

/\*no terminal node entradaESaida.js -a --> Gera uma flag para o arquivo

stdout - standard out (Saída padrão)

\*/

const anonimo = process.argv.indexOf('-a') !== -1

console.log(anonimo)

if (anonimo) {

process.stdout.write('Fala anônimo\n')

process.exit()

} else {

process.stdout.write('Informe seu nome:')

process.stdin.on('data', data =>{

const nome = data.toString().replace('\n', '')

process.stdout.write(`Fala ${nome}\n`) //Disparado sempre que o usuário tecla enter

process.exit()

})

}

# ES Next... 6,7,8

## Revisão #01

/\*

var: escopo de função ou global

let: escopo de bloco, alem de escopo de função e global

\*/

{

var a = 2

let b = 3

console.log(b)

}

console.log(a) //Acessa

//console.log(b) //Não acessa, definida apenas no bloco

//Template String

const produto = 'iPad'

console.log(`${produto} é caro`)

//Destructing

//---String

const [l, e, ...tras] = 'Cod3r'

console.log(l, e, tras)

//---Array

const [x, , z] = [1, 2, 3]

console.log(x, z)

//-- Object

const { idade: i, nome: n } = { nome: 'Liana', idade: 27 }

console.log(`Idade: ${i}\nNome: ${n}`)

## Revisão #02

## Revisão #03

## Tagged Template

# Fundamentos da Web

# HTML

# CSS

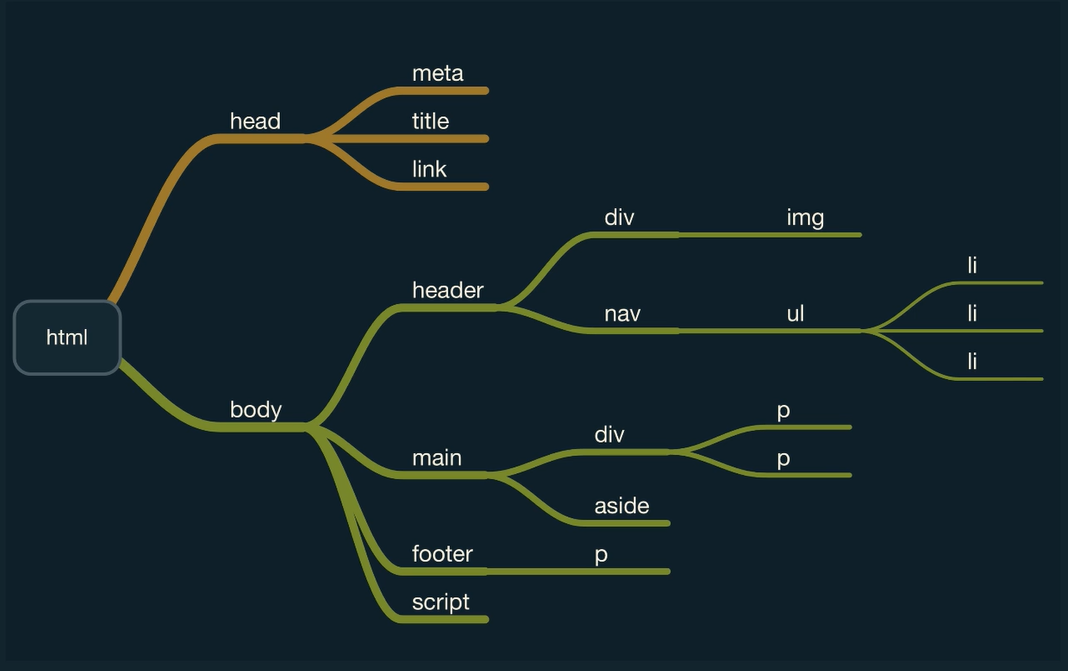
# Integração HTML, CSS e JavaScript

## DOM

Dom significa ‘Document object model’, e representa a estrutura do documento html e o relacionamento entre os elementos, as tags (Figura 2).

A dom é essa estrutura em arvore que pega a sua página html (texto) e gera essa estrutura em arvore. Podemos ler e manipular a dom.

Figura 2. Representação gráfica do dom.



# React

## Criando o Projeto

Tecnicamente o react é uma biblioteca, que trabalha apenas com a camada de visão (view) dos componentes.

Para iniciar um projeto react, precisamos ir à pasta e digitar 'npm i -g create-react-app' e criar o app, já com a build configurada. Por baixo, o react um build do webpack.

Para criar, basta entrar com 'create-react-app exercicios' onde exercicios é o nome do projeto.

O elemento onde nossa aplicação vai ser ‘colocada’ dentro, é o elemento cujo id é ‘root’.

Passar uma tag por parâmetros de uma função, não é sintaxe JavaScript, essa sintaxe é de uma ‘extensão’ js, o jsx. Essa extensão, vai permitir programar mais parecido com html, e na execução, vai ter uma conversão para JavaScript, assim como ocorre com TypeScript. Por baixo dos panos serão realizadas várias chamadas para conversão para JavaScript.

## Componentes Baseados em Função

ReactDom.render(**<h1>React</h1>**, document.getElementById('root'))

Todos os componentes (Primeiro) precisam ter a primeira letra maiúscula.

**Em Primeiro.jsx:**

import React from 'react'

export default function h1() {

    return <h1>React</h1> //Todas tags precisam ser fechadas (mesmo se não precisar)

}

**Em index.js:**

import React from 'react'

import ReactDom from 'react-dom'

import **P**rimeiro from './componentes/Primeiro'

ReactDom.render(<*Primeiro* />, document.getElementById('root'))

Não podemos renderizar duas tags (Ex: uma h1 e outra h2) em um mesmo componente, para isso, precisamos envolvê-las por uma div (Configurando uma única tag), usar o React.Fragment ou exportar um array com as tags nas posições; nesse caso, precisamos passar uma chave como propriedade, para o react encontrar de forma mais otimizada os elementos dentro da dom.

export default *props* =>

    <div>

        <h1>{*props*.nome}!</h1>

        <h2>Até breve</h2>

    </div>

Ou

export default *props* =>

    <*Fragment*>

        <h1>{*props*.nome}!</h1>

        <h2>Até breve</h2>

    </*Fragment*>

Ou

export default props => [

        <h1 key="h1">{props.nome}!</h1>,

        <h2 key="h2">Até breve</h2>

    ]

Ao exportar um componente, se não usarmos o default, precisamos desestruturá-lo na hora de importar.

**Em index.js:**

import React from 'react'

import ReactDom from 'react-dom'

import { BoaTarde, BoaNoite} from './componentes/Multiplos'

ReactDom.render(

    <div>

        <BoaTarde nome = "Lucas"/>

        <BoaNoite nome = "Liana"/>

    </div>

, document.getElementById('root'))

**Em Multiplos.jsx:**

import React from 'react'

// eslint-disable-next-line

export const BoaTarde = props => <h1>Boa Tarde {props.nome}</h1>

export const BoaNoite = props => <h1>Boa Noite {props.nome}</h1>

Ou

**Em index.js:**

import React from 'react'

import ReactDom from 'react-dom'

import Multi from './componentes/Multiplos'

ReactDom.render(

    <div>

        <Multi.BoaTarde nome = "Lucas"/>

        <Multi.BoaNoite nome = "Liana"/>

    </div>

, document.getElementById('root'))

**Em Multiplos.jsx:**

import React from 'react'

export const BoaTarde = props => <h1>Boa Tarde {props.nome}</h1>

export const BoaNoite = props => <h1>Boa Noite {props.nome}</h1>

export default { BoaTarde, BoaNoite}

## Componentes Baseados em Classes

Os componentes baseados em classes possuem algumas funcionalidades que os componentes baseados em funções não possuem, como a capacidade de trabalhar com estados (verificar, talvez hoje tenha com o ‘hook’). E os métodos de ciclo de vida, ou seja, chamar um determinado trecho de código quando um elemento for renderizado, destruído etc.

Ao chamar: <Saudacao tipo="Bom dia" nome="Liana"/>, estamos instanciando a classe ‘Saudacao’, e os valores dos parâmetros (tipo e nome), que estão em ‘this.props’ (no caso de classes), são somente leitura, ou seja, não conseguimos alterar as propriedades dos componentes. Para alterar esses parâmetros através de um input, precisamos trabalhar com state.

Dentro de todo componente de classe, tem um atributo chamado state,

Que vai permitir que alteremos os parâmetros do nosso componente.

‘Quando o estado do componente modifica, aí que o componente é atualizado (visual).

Em algumas versões do Angular, tínhamos um formato de atualização ‘to way data bind’, onde o componente(visual) pode atualizar os dados, quanto os dados podem atualizar os componentes. Em React isso não existe, o fluxo de atualização é sempre unidirecional, sempre o estado mudo para atualizar o componente. Um evento é disparado, que atualizar o estado, aí sim, modifica o componente (componente controlado). O dado é quem manda na renderização do componente.

Chamar o evento assim ‘value={tipo} onChange={this.setTipo}/>’, passando a referência da função, teremos problemas, pois o this não aponta para o componente atual, para isso, precisamos seta-lo com o bind:

import React, {Component} from 'react'

// eslint-disable-next-line

export default class Saudacao extends Component {

    state = {

        tipo: this.props.tipo,

        nome: this.props.nome

    }

    // eslint-disable-next-line

    constructor (props) {

        super(props) //super = Component

        this.setTipo = this.setTipo.bind(this) //Dentro de bind, o this vai ser o componente atual, pois dentro do construtor, o this é uma instância de Saudacao

        this.setNome = this.setNome.bind(this)

    }

    setTipo(e) {

        this.setState({ tipo: e.target.value }) //Passamos um objeto com os parâmetros que queremos alterar

    }

    setNome(e) {

        this.setState({ nome: e.target.value }) //Passamos um objeto com os parâmetros que queremos alterar

    }

    render(){

        const { tipo, nome } = this.state

        return (

            <div>

                <h1>{tipo} {nome}!</h1>

                <input type="text" placeholder="Tipo..."

                value={tipo} onChange={this.setTipo}/>

                <input type="text" placeholder="Nome..."

                value={nome} onChange={this.setNome}/>

            </div>

        )

    }

}

Outra forma, será chamar com uma função arrow:

import React, {Component} from 'react'

// eslint-disable-next-line

export default class Saudacao extends Component {

    state = {

        tipo: this.props.tipo,

        nome: this.props.nome

    }

    setTipo(e) {

        this.setState({ tipo: e.target.value }) //Passamos um objeto com os parâmetros que queremos alterar

    }

    setNome(e) {

        this.setState({ nome: e.target.value }) //Passamos um objeto com os parâmetros que queremos alterar

    }

    render(){

        const { tipo, nome } = this.state

        return (

            <div>

                <h1>{tipo} {nome}!</h1>

                <input type="text" placeholder="Tipo..."

                value={tipo} onChange={e => this.setTipo(e)}/>

                <input type="text" placeholder="Nome..."

                value={nome} onChange={e => this.setNome(e)}/>

            </div>

        )

    }

}

Componentes podem herdar propriedades de outros, através do comando spread

**Em index.js:**

import React from 'react'

import ReactDom from 'react-dom'

import Pai from './componentes/Pai'

ReactDom.render(

    <div>

        <Pai nome="Lucas" sobrenome="Peixoto" />

    </div>

    , document.getElementById('root'))

**Em Pai.jsx:**

import React from 'react'

import Filho from './Filho'

// eslint-disable-next-line

export default props =>

    <div>

        <h1>{props.nome} {props.sobrenome}</h1>

        <h2>Filhos</h2>

        <ul>

            <Filho nome="César" sobrenome={props.sobrenome} />

            <Filho {...props} />

            <Filho {...props} nome="Cecília" /> {/\* Esse filho herdou todas as props do Pai, se quiser alterar alguma, tem que vim depois \*/}

        </ul>

    </div>

**Em Filho.jsx:**

import React from 'react'

// eslint-disable-next-line

export default props => <li>{props.nome} {props.sobrenome}</li>

Podemos também, incluir os filhos na própria chamada do componente Pai:

**Em index.js:**

import React from 'react'

import ReactDom from 'react-dom'

import Pai from './componentes/Pai'

import Filho from './componentes/Filho'

ReactDom.render(

    <div>

        <Pai nome="Lucas" sobrenome="Fernandes Peixoto" />

            <Filho nome="Cecília" sobrenome="Fernandes Peixoto" />

            <Filho nome="César" sobrenome="Fernandes Peixoto" />

    </div>

    , document.getElementById('root'))

**Em Pai.jsx:**

import React from 'react'

import Filho from './Filho'

// eslint-disable-next-line

export default props =>

    <div>

        <h1>{props.nome} {props.sobrenome}</h1>

        <h2>Filhos</h2>

        <ul>

           {props.children}

        </ul>

    </div>

E se quisermos aproveitar o sobre nome em index.js para colocar nos filhos:

# Ajax

# Gulp

# Webpack

# jQuery

# Bootstrap

# Vue

# Banco de Dados Relacionais

# Banco de Dados Não Relacionais

# Express

# Mongoose

# Projetos

# Outros Tópicos

# Conclusão