**JAVASCRIPT**

**for in:** retornar as chaves e os respectivos valores do objeto.

Ex: Puxar um relatório com as informações do cliente (ex: nome = Caio, idade = 26). Detalhe que o objeto possui diversos tipos de informações como number, string, arrays, object, functions, porém não queremos todo tipo de informação, sendo necessário excluir as indesejadas.

const cliente = {

    nome: 'Caio',

    idade: 26,

    CPF: '00011122299',

    email: "caiomiranda701@hotmail.com",

    telefone: ['81999006455', '81993016455'],

    dependentes: [

        {

        nome: 'Niedja',

        parentesco: 'noiva',

        dataNasc: '01/01/2000'},

        {

        nome:'Rodrigo',

        parentesco: 'irmao',

        dataNasc: '01/01/2010'

        }

    ],

    saldo: 100,

    depositar:function(valor){

        this.saldo += valor;

    }

}

let relatorio = '';

for (let info in cliente) {

    if (typeof cliente[info] === 'object' || typeof cliente[info] === 'function') {

        continue

    } else {

        relatorio += `

        ${info} = ${cliente[info]}

        `

    }

}

console.log(relatorio);

**var.includes(valor)** Verifica se determinado valor está incluído dentro do objeto. OBS: *includes* retorna um valor true ou false. Geralmente utilizado no *if.*

**Object.keys(objeto);** pede para o JS extrair do objeto uma lista de chaves.

* **Object.values(var):** pede para o JS extrair do objeto uma lista com os valores das chaves.
* **Object.entries(var):** pede para o JS extrair do objeto uma lista com as chaves e seus respectivos valores. OBS: Retorna em uma estrutura diferente, trazendo cada chave e respectivo valor dentro de um *array* (índice 0 se referindo a chave e índice 1 ao valor).

Ex: Um banco quer oferecer seguro para o cliente que possuir dependentes em seu cadastro. Então será necessário verificar no objeto “cliente”, que é onde se encontra as informações do cliente, e ver se há algum dependente. Primeiro utiliza o *object.keys* para puxar todas as chaves do objeto “cliente” e depois utiliza o *Object.include* em um *if* para verificar se há algum dependente.

const cliente = {

    nome: 'Caio',

    idade: 26,

    CPF: '00011122299',

    email: "caiomiranda701@hotmail.com",

    telefone: ['81999006455', '81993016455'],

    dependentes: [

        {

        nome: 'Niedja',

        parentesco: 'noiva',

        dataNasc: '01/01/2000' },

        {

        nome:'Rodrigo',

        parentesco: 'irmao',

        dataNasc: '01/01/2010'

        }

    ],

    saldo: 100,

    depositar:function(valor){

        this.saldo += valor;

    }

}

function oferecerSeguro (objeto) {

    const propsClientes = Object.keys(cliente);

    if (propsClientes.includes('dependentes')){

        console.log(`Oferta de seguro disponpivel para ${objeto.nome}`);

    }

}

oferecerSeguro (cliente);

**operador de espalhamento (...)** também chamado de *spread operator*, tem o intuito de “espalhar” as informações de um *array* de objetos em um outro *array*.

**console.table(var);** mostra o resultado em um formato de tabela.

Ex: O banco quer extrair de uma listagem de clientes somente as informações dos dependentes e montar uma única lista para análise de outro departamento. Sendo assim, será necessário acessar informações dos dependentes de dentro dos clientes (*array* dentro de um *array*) e posteriormente inserir em um outro *array*. Por isso utiliza-se o “...” para poder inserir o conjunto de informações dos dependentes dentro de um *array*.

const clientes = [

    {

      nome: 'Caio',

      CPF: '00011122299',

      dependentes: [{

        nome: 'Niedja',

        parentesco: 'noiva',

        dataNasc: '01/01/2000' },

        {

        nome:'Rodrigo',

        parentesco: 'irmao',

        dataNasc: '01/01/2010'}],

    },

    {

      nome: 'Juliana',

      CPF: '00011122288',

      dependentes: [{

        nome: 'Andre',

        parentesco: 'marido',

        dataNasc: '01/01/2005' }],

    }

]

let listaDependentes = [];

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

  listaDependentes.push(...clientes[i].dependentes);

}

console.table(listaDependentes);

//

┌─────────┬───────────┬────────────┬──────────────┐

│ (index) │ nome │ parentesco │ dataNasc │

├─────────┼───────────┼────────────┼──────────────┤

│ 0 │ 'Niedja' │ 'noiva' │ '01/01/2000' │

│ 1 │ 'Rodrigo' │ 'irmao' │ '01/01/2010' │

│ 2 │ 'Andre' │ 'marido' │ '01/01/2005' │

└─────────┴───────────┴────────────┴──────────────┘ //

**funcao.call(parâmetros);** para chamar as propriedades de uma função para outra

**.prototype** é utilizado para o JS acessar um objeto e entrar na propriedade prototype, adicionando um novo método na função.

Ex: O banco quer aproveitar as informações da conta do cliente para criar uma nova conta (conta poupança) e que seja possível realizar depósitos nessa conta. Sendo assim, primeiro é necessário criar uma função genérica à conta do cliente (utilizando o *call*), no caso para a conta poupança, e além das informações já existentes da conta, acrescentar a informação de saldo da conta poupança. Em seguida, é necessário utilizar o *prototype* para acessar a propriedade do objeto e criar um novo método (para que seja possível realizar depósitos na conta poupança).

function Cliente(nome, cpf, email, saldo){

    this.nome=nome

    this.cpf=cpf

    this.email=email

    this.saldo=saldo

    this.depositar = function(valor){

        saldo += valor;

    }

}

function ClientePoupanca(nome, cpf, email, saldo, saldoPoupanca){

    Cliente.call(this, nome, cpf, email, saldo)

    this.saldoPoupanca = saldoPoupanca

}

const ju = new ClientePoupanca ('Ju', '00011122233', 'ju@email.com', 100, 200);

console.log(ju);

ClientePoupanca.prototype.depositarPoupanca = function(valor){

    this.saldoPoupanca += valor;

}

ju.depositarPoupanca(30);

console.log(ju.saldoPoupanca);

// ClientePoupanca {

nome: 'Ju',

cpf: '00011122233',

email: 'ju@email.com',

saldo: 100,

depositar: [Function (anonymous)],

saldoPoupanca: 200

}

230 //

**Classes** são modelos usados para criação de objetos. Ela define os atributos (características) e os métodos (comportamentos) de um objeto. OBS1: Sempre damos o nome da *class* com a primeira letra maiúscula. OBS2: Dentro da *class*, ao declarar uma função, não é necessário escrever a palavra *function*.

* **Instancias** são objetos criados a partir de uma classe.

**Constructor** é uma função na qual você chamará no código posteriormente através do *new*.

**Herança** é um mecanismo da orientação a objetos que permite que uma classe herde características de uma outra classe.

**Objeto** é uma instancia de classe em memória.

* **Métodos** são funções executadas dentro do objeto.

class Cliente{

    constructor (nome, cpf, email, saldo){

        this.nome = nome

        this. cpf = cpf

        this.email = email

        this.saldo = saldo

    }

    depositar(valor){

        this.saldo += valor

    }

    exibirSaldo(){

        console.log(this.saldo);

    }

}

const caio = new Cliente ('Caio', '00011122233', 'caio@email.com', 100);

caio.exibirSaldo();

caio.depositar(30);

console.log(caio);

/\*

100

  Cliente {

    nome: 'Caio',

    cpf: '00011122233',

    email: 'caio@email.com',

    saldo: 130} \*/

**this** representa o objeto que executa a função. No caso de atributos das funções construtoras ou dos construtores de classe, o *this* tem função similar, fazendo com que o construtor receba os atributos deste objeto. Ele vai se refefir a instância e pode acessar suas propriedades. Caso esteja fora do escopo local, irá se refrir ao objeto global (*window* no *devtools* do chrome; *global* no NodeJs).

//o this quando inserido num objeto, aponta para o objeto. Nesse caso o this está apontando para o obj e através do "." está acessando a propriedade "prop" do objeto.

const obj = {

    prop: 100,

    f: function() {

        return this.prop;

    }

};

console.log(obj.f());

// 100

Quando estamos trabalhando com classes e instancias (objetos criados através da class) e com arrow function (this dentro da arrow function):

// Aqui temos uma classe que recebe 1 método (thisEhIgualObj) e dentro desse método, recebe outro método (a setTimeout) em modo arrow function.

// Estamos querendo saber se, no primeiro método, o this está se referindo a instância obj (objeto derivado da classe). Conclui-se que é true porque o this está dentro do escopo atual (dentro do escopo da instância obj).

// No segundo método (que está dentro do método anterior), queremos saber se o this está se referindo a instância obj, e concluimos que é true. O que acontece é que, como foi passado em arrow function, o this se refere ao escopo "mãe", ou seja, o escopo acima do que ele se encontra (no caso, o escopo fora do método que ele está dentro, sendo o escopo do obj). Se esse método estivesse no escopo obj (ao invés de estar dentro do método thisEhIgualObj), ele seria dado com undefined, ou global, visto que estaria apontando para o escopo global (pois é o escopo acima do obj).

class MyClass {

    constructor(valor){

        this.prop1 = valor;

}

    thisEhIgualObj () {

        console.log(this === obj)

        setTimeout (() => {

            console.log(this===obj)

        }, 500)

    }

}

const obj = new MyClass('valor')

obj.thisEhIgualObj()

//true

//true

É possível manipular os valores do *this* através dos métodos:

* **Call( )** permite que uma função seja chamada com parâmetros e valor de *this* especifico. No exemplo abaixo, o primeiro parâmetro do método *call* se refere ao objeto que será usado como contexto do *this*, e do segundo parâmetro em diante, são passados os argumentos que a função deve receber. No caso, há somente um parâmetro (tipoCliente).
* function imprimeNomeEmail(tipoCliente){
* console.log(`${tipoCliente} - nome: ${this.nome}, emmail: ${this.email}`)
* }
* const cliente1 = {
* nome: 'Carlos',
* email:'c@email.com'
* }
* const cliente2 = {
* nome: 'Fred',
* email:'f@email.com'
* }
* // No caso iremos pegar a função e aplicar para as novas constantes. Primeiro será aplicada a função para o cliente 1, sendo atribuído como parâmetro o ‘cliente especial’ e depois será aplicada a mesma função, porém para o cliente 2, utilizando como parâmetro ‘cliente estudante’.
* imprimeNomeEmail.call(cliente1, 'cliente especial');
* imprimeNomeEmail.call(cliente2, 'cliente estudante');
* /\* cliente especial - nome: Carlos, emmail: c@email.com
* cliente estudante - nome: Fred, emmail: f@email.com
* \*/
* **Bind( )** cria uma nova função que, quando chamada, tem a palavra chave *this* definida com o valor fornecido. O *bind* “prende” uma função ao contexto de um objeto.
* const personagem = {
* nome: "Princesa Leia",
* apresentar: function(){
* return `a personagem é ${this.nome}`
* }
* }
* const personagemGenerico = personagem.apresentar
* // No caso, criamos uma variável que chama o objeto com sua propriedade = personagem.apresentar. Porém precisamos acrescentar o bind juntamente com objeto ou parâmetro ao qual queremos fixar o this = bind(personagem). Pois, caso não acrescentemos o bind(personagem), o “this” não vai ter referência, visto que foi atribuído a uma nova variável
* const personagemDefinido = personagemGenerico.bind(personagem)
* console.log(personagemDefinido());
* //a personagem é Princesa Leia
* **Aplly( )** funciona de forma semelhante ao *call* porém recebe os argumentos em um *array*. Atenção para que o *array* siga a ordem dos parâmetros informados na função.
* function imprimeNomeEmail(tipoCliente, agencia){
* console.log(`${tipoCliente} da agência ${agencia}: - nome: ${this.nome}, email: ${this.email}`)
* }
* const cliente1 = {
* nome: "Carlos",
* email: "c@email.com"
* }
* const cliente2 = {
* nome: "Fred",
* email: "f@email.com"
* }
* const clienteEspecial = ['cliente especial', 'Rio de Janeiro'];
* const clienteEstudante = ['cliente estudante', 'Fortaleza'];
* // Nesse caso, queremos fazer a mesma coisa do call, porém temos mais de 1 parâmetro a ser recebido, sendo utilizado um array.
* imprimeNomeEmail.apply(cliente1, clienteEspecial);
* imprimeNomeEmail.apply(cliente2, clienteEstudante);
* /\*cliente especial da agência Rio de Janeiro: - nome: Carlos, email: c@email.com
* cliente estudante da agência Fortaleza: - nome: Fred, email: f@email.com \*/

**extends** é utilizado quando você quer herdar atributos de uma *class* existente para uma nova *class*.

**super** é utilizado para passar as propriedades que você quer herdar.

class Cliente {

  constructor(nome, email, cpf, saldo){

    this.nome = nome;

    this.email = email;

    this.cpf = cpf;

    this.saldo = saldo;

  }

  depositar(valor){

    this.saldo += valor;

  }

  exibirSaldo(){

    console.log(this.saldo);

  }

}

class ClientePoupanca extends Cliente {

  constructor(nome, email, cpf, saldo, saldoPoupanca){

    super (nome, email, cpf, saldo);

    this.saldoPoupanca = saldoPoupanca;

  }

  depositarPoupanca(valor){

    this.saldoPoupanca += valor;

  }

}

const caio = new ClientePoupanca ('Caio', 'c@email.com', '00011122233', 100, 200);

caio.depositar(50);

caio.depositarPoupanca(100);

console.log(caio);

/\* ClientePoupanca {

  nome: 'Caio',

  email: 'c@email.com',

  cpf: '00011122233',

  saldo: 150,

  saldoPoupanca: 300 } \*/

**Object.setPrototypeOf(parametro1, parametro2);** para utilizar propriedades de um objeto em outro. Parametro1 é o objeto que irá utilizar a propriedade herdada e parametro2 é o objeto que irá herdar a propriedade.

const user = {

    nome: 'Caio',

    email: 'c@email.com',

    nascimento: '12/02/1996',

    role: 'estudante',

    ativo: true,

    exibirInfos: function () {

        console.log(this.nome, this.email)

    }

}

const admin = {

    nome: 'Mariana',

    email: 'm@m.com',

    role: 'admin',

    criarCurso() {

        console.log('curso criado');

    }

}

Object.setPrototypeOf(admin, user);

admin.criarCurso();

admin.exibirInfos();

/\* curso criado

Mariana m@m.com \*/

**Object.creat(parametro);** para criar um objeto, de maneira similar ao *new*.

**Object.freeze(parametro);** congela um objeto com todas as suas propriedades. Qualquer alteração de sua propriedade será ignorada. Podemos aplica-lo no construtor da classe, tendo como alvo o *this*, mas só depois de termos atribuído os valores recebidos no construtor às propriedades.

**package.json** arquivo necessário para importar e exportar blocos de códigos para outros códigos. Ex: quando se quer trabalhar com herança de classes, aproveitando métodos de outra classe que está em outro arquivo. Para criar o arquivo packeg.jason:

1. No terminal escreva *npm init* e depois aperte *enter*
2. Em seguida será necessário inserir algumas informações básicas no terminal para ser criado o arquivo como (nome, descrição, versão, autor etc). Não é necessário escrever em todos os tópicos (quando não quiser escrever nada, basta apertar *enter*.
3. Após inserir as informações requisitadas, aperte *enter* na última opção (YES) para então ser criado o arquivo na pasta.
4. No arquivo package.json, em qualquer parte (abra uma linha entre as propriedades), escreva “type”: “module”,

**get** é uma propriedade “assessora” (assessor), na qual recebe um valor (return) e faz a leitura guardando a informação. Atuam na segurança de uma classe, encapsulando dados quando necessário. Esse assessor não aceita nenhum parâmetro.

**set** é usado para definir/alterar propriedades. É útil quando se deseja executar algum código sempre que alguma propriedade é definida ou sofre alguma modificação. Precisa receber um parâmetro. Ex: códigos que façam validação de campos. Esse assessor só aceita um parâmetro.

**Polimorfismo** é o princípio segundo o qual duas ou mais classes que herdam de uma superclasse podem executar métodos vindos dessa superclasse e que tem a mesma assinatura, porém comportamentos diferentes da superclasse da qual herdam.

**Encapsulamento** é o conceito de “esconder” propriedades e métodos para que não possam ser acessados por fora da classe, o que pode ser feito com o uso de atributos privados e propriedades *assessors*.

**Throw new Error(“texto”)** faz com que seja lançado um erro, impedindo assim a execução da tarefa (impedir a criação de uma classe por exemplo).

**Classe abstrata** é uma classe que não pode ser instanciada, podendo somente ser herdada.

**Assíncrono** quer dizer que não acontece exatamente no momento (função *callback* é assíncrona geralmente).

**getDate( );** retorna os dias úteis da semana. Começando com o 0 (domingo) até o número 6 (sábado).

**forEach( )** executa uma dada função em cada elemento de um array.

Array.forEach(callback( ) => { }

A função de callback que passamos pode receber até 3 parâmetros (não é necessário utilizar todos os 3):

1. O item atual que está sendo iterado (o item individual da array);
2. O index do item atua;
3. O array que está sendo iterado;

Uma vantagem do *forEach* é que a função pode ser desacoplada, deixando o código, em alguns casos, mais organizado e legível.

const nomes = ['caio', 'rodrigo', 'niedja'];

nomes.forEach(function(nomePorPessoa, index, array){

    console.log(nomePorPessoa, index, array);

})

// caio 0 [ 'caio', 'rodrigo', 'niedja' ]

// rodrigo 1 [ 'caio', 'rodrigo', 'niedja' ]

// niedja 2 [ 'caio', 'rodrigo', 'niedja' ]

No modo *arrow function*, fica:

const nomes = ['caio', 'rodrigo', 'niedja'];

nomes.forEach((nome, index, array)=>{

    console.log(nome, index, array);

})

// caio 0 [ 'caio', 'rodrigo', 'niedja' ]

// rodrigo 1 [ 'caio', 'rodrigo', 'niedja' ]

// niedja 2 [ 'caio', 'rodrigo', 'niedja' ]

Desacoplando a função, fica:

const nomes = ['twitter', 'instagram', 'facebook'];

const funcaoDesacoplada = (redeSocial, indice, arrayCompleto)=>{

    console.log(redeSocial, indice, arrayCompleto); }

nomes.forEach(funcaoDesacoplada);

// twitter 0 [ 'twitter', 'instagram', 'facebook' ]

// instagram 1 [ 'twitter', 'instagram', 'facebook' ]

// facebook 2 [ 'twitter', 'instagram', 'facebook' ]

Exemplo prático e trabalhando com objetos na *array*: Calcular desconto do produto e colocar o desconto em cada produto.

­­ // Calcular desconto do produto e colocar o desconto em cada produto.

let produtos = [

    {

        nomeProduto: 'Celular',

        valorProduto: 1200,

        porcentagemDesconto: 0.2

    },

    {

        nomeProduto: 'Notebook',

        valorProduto: 2500,

        porcentagemDesconto: 0.3

    },

    {

        nomeProduto: 'Televisao',

        valorProduto: 35000,

        porcentagemDesconto: 0.35

    }

];

produtos.forEach((produtoAtual)=>{

    let valorDesconto = produtoAtual.valorProduto \* produtoAtual.porcentagemDesconto;

    let valorFinal = produtoAtual.valorProduto - valorDesconto;

    produtoAtual.valorTotalProduto = valorFinal

console.log(produtoAtual);

})

/\*

{

  nomeProduto: 'Celular',

  valorProduto: 1200,

  porcentagemDesconto: 0.2,

  valorTotalProduto: 960

}

{

  nomeProduto: 'Notebook',

  valorProduto: 2500,

  porcentagemDesconto: 0.3,

  valorTotalProduto: 1750

}

{

  nomeProduto: 'Televisao',

  valorProduto: 35000,

  porcentagemDesconto: 0.35,

  valorTotalProduto: 22750

} \*/

**Map( )** é um método que invoca a função *callback* passada por argumento para cada elemento do *array* e devolve um novo *array* como resultado.

Array.map(function(elemento){ })

const nomes = ['ayrton', 'niedja', 'caio'];

const nomesMaiusculos = nomes.map(function(nome){

    return nome.toUpperCase();

})

console.log(nomesMaiusculos);

// ['AYRTON', 'NIEDJA', 'CAIO']

Em *arrow function*, fica:

const nomes = ['ayrton', 'niedja', 'caio'];

const nomesMaiusculos = nomes.map(nome =>nome.toUpperCase());

console.log(nomesMaiusculos);

// ['AYRTON', 'NIEDJA', 'CAIO']

**Filter( )** é um método que cria um novo *array* com todos os elementos que passaram no teste implementado pela função fornecida. Retorna um booleano, no qual, se retornar *true*, a cópia do item que está sendo verificado é adicionado ao novo array que está sendo criado.

Array.filter((item, index, array) => { })

Somente o primeiro parâmetro (item) é obrigatório.

Exemplo: criar uma lista somente com os números maiores que 37.

const numeros = [35, 36, 37, 38, 50, 55, 90];

const numerosMaiores = numeros.filter(numero => numero > 37);

console.log(numerosMaiores);

// [ 38, 50, 55, 90 ]

Exemplo: Eliminando itens duplicado, criando um novo array.

const numerosOriginais = [1, 0, 3, 4, 8, 4, 3, 1, 9, 8, 7];

const numerosNovos = numerosOriginais.filter((item, indice, array) => array.indexOf(item) === indice);

console.log(numerosOriginais);

console.log(numerosNovos);

/\* [ 1, 0, 3, 4, 8, 4, 3, 1, 9, 8, 7 ]

[ 1, 0, 3, 4, 8, 9, 7 ] \*/

**Reduce( )** é um método que executa uma função “reducer” (fornecida por mim) para cada elemento do array, resultando num único valor de retorno.

Array.reduce(callback(acumulador, valorAtual) { }

* acumulador é o valor acumulado que está sendo somado e será mostrado no fim.
* valorAtual é o item atual da array que será somado, sendo acrescentado ao acumulador.
* const idades = [8, 13, 27, 39, 40, 22];
* const somaIdades = idades.reduce(function(valorAcumulador, valorArrayAtual) {
* return valorAcumulador + valorArrayAtual; }, 0);
* // valorAcumulador é o valor que ficará sendo acumulado até o resultado final.
* // valorArrayAtual é o valor atual que será iterado ao valor total (nesse caso será somado ao valorAcumulador.
* // o 0 no final é o valor inicial e o tipo que queremos retornar (escolhemos começar com 0 pois queremos um número e que comece com 0)
* console.log(somaIdades);
* // 149

No modo *Arrow Function*, fica:

const numeros = [8, 13, 27, 39, 40, 22];

const somaNumeros = numeros.reduce((valorAcum, valorAtual) => valorAcum + valorAtual, 0);

console.log(somaNumeros);

//149

Exemplo prático: Transformar um array de objetos com nome e idade das pessoas, organizando em um objeto de maneira destrinchada entre as pessoas maiores de idade e as menores de idade.

const pessoas = [   { nome: 'Ayrton', idade: 17 },

                    { nome: 'niedja', idade: 25 },

                    { nome: 'caio', idade: 26 },

                    { nome: 'rodrigo', idade: 18 },

                    { nome: 'claudia', idade: 15 } ];

const pessoasAgrupadas = pessoas.reduce(function(acumulador, atual) {

    const maiorOuMenor = atual.idade >= 18 ? 'maiores' : 'menores';

    acumulador[maiorOuMenor].push(atual);

    return acumulador; }, { maiores: [], menores: []})

    console.log ('pessoasAgrupadas', pessoasAgrupadas);

/\* pessoasAgrupadas { maiores: [ { nome: 'niedja', idade: 25 }, { nome: 'caio', idade: 26 }, { nome: 'rodrigo', idade: 18 } ],

  menores: [ { nome: 'Ayrton', idade: 17 }, { nome: 'claudia', idade: 15 } ] } \*/

**Arrow function:**

Um exemplo de função, escrita da maneira comum:

Function apresentar (nome) {

Return `meu nome é ${nome}`; }

Utilizando no modelo Arrow Function, fica:

const apresentarArrow = nome => `meu nome é ${nome}`;

Outro exemplo de Arrow Function.

const soma = (num1, num2) => num1 + num2;

Um exemplo de Arrow Function com mais de 1 instrução:

const somaNumerosPequenos = (num1, num2) => {

if (num1 || num2 > 10) {

return “somente números de 1 a 9” }

else {

return num1 + num2; } }

OBS: Quando a função possui somente uma instrução, não é necessário utilizar as chaves { }

OBS: Quando utilizar as chaves, é necessário inserir o *return* dentro da função.

Exemplo sem return:

const sum = (number1, number2) => number1 + number2;

console.log(sum(20, 10));

//30

Exemplo sem parâmetro:

const comunidade = () => 'discord';

console.log(comunidade());

// discord

Exemplo somente com 1 parâmetro: Não precisa do ( )

const double = numero => numero \* 2;

console.log (double(3));

// 6

Exemplo JSON (objeto literal) sem *return*:

const getPerson = () => ({nome: 'Niedja', idade: 25});

console.log(getPerson());

// { nome: 'Niedja', idade: 25 }

**Callback** são funções que possuem uma chamada de retorno. Aguardam um evento para ser executada.

// Primeiro você declara uma variável, atribuindo uma função como parâmetro (será a função callback).

const minhaFuncao = funcaoCallback => {

    const valor = 77;

    funcaoCallback(valor);

}

// Em seguida você chama a variável, atribuindo uma nova função como parâmetro, e a mesma irá inserir a função callback dentro dela.

minhaFuncao (numero => {

    console.log(numero);

})

// 77

Outro exemplo:

function soma (a, b, callback){

    var op = a + b;

    callback(op);

}

function multiplica (a, b, cb){

    var op = a \* b;

    cb(op);

}

function exibir (numero) {

    console.log ("A operação resultou em: " + numero);

}

soma (2, 3, exibir);

multiplica(4, 5, exibir);

// A operação resultou em: 5

// A operação resultou em: 20

**Instanceof** é utilizado para saber se a variável é uma instancia do construtor, classe, função, etc.

var foo = new Number (123);

var bar = Number (345);

console.log(foo instanceof Number);

console.log(bar instanceof Number);

//true (retorna true porque foo é uma instância de Number, ou seja, é um objeto literal da class Number)

//false (retorna false porque bar não é um objeto literal de Number, ou seja, não foi criado um novo objeto (uma instância) a partir da classe Number).

**indexOf( )** é o método que retorna o primeiro índice em que o elemento pode ser encontrado no array; Retorna -1 caso o mesmo não esteja presente.

let frutas = ['pera', 'uva', 'laranja'];

console.log(frutas.indexOf('uva'));

// 1 (retorna o índice do array que está a uva)

Exemplo com duas palavras repetidas na *string*, e querendo buscar a segunda palavra.

let texto = 'um texto para teste, vamos ver se o teste da certo';

console.log(texto.indexOf('teste', 15));

// 36 (retorna o índice 36, que é quando começa a segunda palavra teste, visto que solicitamos para começar a busca somente a partir do índice 15, que é quando o início da primeira palavra "teste" já passou).

const numerosOriginais = [1, 0, 3, 4, 8, 4, 3, 1, 9, 8, 7];

const numerosNovos = numerosOriginais.filter((item, indice, array) => array.indexOf(item) === indice);

console.log(numerosOriginais);

console.log(numerosNovos);

/\* [ 1, 0, 3, 4, 8, 4, 3, 1, 9, 8, 7 ]

[ 1, 0, 3, 4, 8, 9, 7 ] \*/

**Regex** são expressões regulares, que são utilizados para identificar determinadas combinações ou cadeias de caracteres em uma string. Através dela podemos validar a entrada de usuários ou encontrar alguma informação em logs, documentação ou saída de comando.

* **Regex-Engine** é o software que interpreta e aplica a regex (Ex: JS, Python, C#...)
* **Meta caracteres** são caracteres que possuem um significado especial para o regex engenie (ou seja, não são interpretados pelo valor literal do caractere).
  + **.** o “ponto” significa qualquer *char* (caractere).
  + **\*** o “asterisco” tem o intuito de definir uma quantidade de chars, zero ou mais vezes
  + **{ }** as “chaves” servem para definir uma quantidade de caracteres específicas que é desejado encontrar.
    - {n} exatamente *n* vezes
    - {n,} no mínimo *n* vezes
    - {n,m} no mínimo *n* vezes e máximo *m* vezes
  + a letra “d” serve para buscarmos dígitos (sem especificar qual).
  + O char “?” serve para informar que o caractere informado na string não é obrigatório (podendo existir ou não). Deve ser passado após a informação do caractere desejado. Informa que o caractere pode estar presente zero ou uma vez.
    - “?:” significa *non-capturing-group* e faz com que não seja capturado a parte informada posteriormente a essa expressão.
    - “+?” faz com que o *regex* pare a busca ao encontro do primeiro caractere informado
  + Utilizamos “( )” para separar as regex em grupos.
  + O char “^” é para informar que é no início da *regex*.
    - O “^” também pode ser usado para informar *tudo que não é.* OBS: Nesse caso, precisa estar dentro de uma classe “[^]
  + O char “$” é para informar que é no final da *regex*.
  + O char [ ] serve para informar que os caracteres passados dentro dele, podem estar presentes ou não na string.
  + \s é o *White space* no qual serve para definir os espaços em branco (ou tabs). Ex: \s+ é para definir que pode ter 1 ou mais espaços em branco. (o símbolo do “+” informa que pode estar presente 1 ou mais vezes)
  + \w significa *word char,* no qual é um atalho para [A-Za-z0-9\_]. Ex: Quando é permitido letras do alfabeto maiúscula ou minúscula e/ou números.
  + \b siginifca *word boundary*, no qual ele verifica se há um *word char* na posição informada (*word char* = \w = [A-Za-z0-9\_]) e caso exista um desses caracteres, ele não considera para a expressão. OBS: \b é um *meta-char* âncora, ou seja, ele marca uma posição específica no alvo. Uma âncora não casa caracteres como as classes fazem, e nem definem quantidades. Existem várias âncoras predefinidas, como “^”, “$” e “\b”.
    - O inverso do “\b”, é o “\B” chamado *non-word-boundary* e tem a função de garantir que há “\w” na posição em que o “\B” for inserido.
  + *Backreferences* são utilizadas quando queremos gerar uma dependência entre os termo. Ex: a tag começa com <h1> e é obrigatório que termine com </h1>. Usa-se um “\” seguido pelo número do grupo, começando com 1.
  + Exemplos: a{3} = letra “a” 3 vezes; \d\* = um digito, zero mais vezes.
  + OBS: Se quisermos procurar por algum caractere especial, porém literalmente o caractere, como por exemplo os “\*”, “.”, “/”, “-“, devemos utilizar o caractere “\”.

Exercícios:

1. Buscar o CNPJ 15.123.321/8883-22.
   1. \d{2}\.\d{3}\.\d{3}\/\d{4}-\d{2}
2. Os IP’s de computadores são formados por 4 grupos de números, em que cada grupo pode conter de 1 a 3 algarismos. Ex: 126.1.111.34; 128.126.12.244; 192.168.0.34.
   1. \d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}
3. Encontre o CEP dentro da frase: João Fulano,123.456.789-00,21 de Maio de 1993,(21) 3079-9987, Rua do Ouvidor,50,20040-030,Rio de Janeiro
   1. \d{5}-\d{3}
4. Encontre o padrão para o número telefônico: (21) 3216-2345. OBS: Considere que o padrão pode conter o novo modelo telefônico com 5 dígitos iniciais: Ex (21) 93216-2345, além disso os parênteses “( )” são *meta-chars*
   1. \(\d{2}\) \d{4,5}-\d{4}
5. Qual a classe correta para definir os números entre 1 e 3 E 6 e 9?
   1. [1-36-9]
6. Como definir uma data no modelo: 28 de Março de 1991 (da maneira mais restrita possível para a quantidade de letras e números).
   1. [0-3]?\d\s+de\s+[A-Z][a-zç]{3,8}\s+de\s+[12]\d{3}
      1. Podemos deixar a expressão mais fácil de entender no JS:

var DIA = “[0-3]?\d”;

var \_DE\_ = “\s+dezs+”;

var MES = “[A-Z][a-zç]{3,8}”;

var ANO = “[12]\d{3}”;

var stringRegex = DIA + \_DE\_ + MES + ANO; (concatenando as variáveis).

var objetoRegex = new RegExp(stringRegex, ‘g’); (passando a string para a regex engine do JS.

1. Qual padrão para definir o horário: 19h32min16s
   1. \d{2}h\d{2}min\d{2}s
2. Validando a placa de um carro. Ex: KMG-8089
   1. [A-Z]{3}-[0-9]{4}
3. Separe os alunos que tiraram notas de 7.2 a 7.9, juntamente com o nome dele, da lista: 9.8 – Robson, 7.1 – Teresa, 4.5 – Armênio, 6.5 – Zulu, 7.7 – Stefania, 7.8 – João, 5.0 – Romeu, 7.2 – Pompilho, 3.1 – Reinaldo, 7.3 – Bernadete, 4.7 – Cinério.
   1. 7\.[2-9]\s+-\s+[^,]+
4. Como selecionar somente as palavras GARROTE, SERROTE E ROTEIRO dentro da lista: GARROTE, BALEIRO, SERROTE, GOLEIRO, ROTEIRO.
   1. [A-Z]\*ROT[A-Z]+
   2. [A-Z]\*ROTE[A-Z]\*
5. Faça uma validação de usuário que possua as seguintes regras: 1. Limite de 10 caracteres; 2. O primeiro caractere deve ser uma letra do alfabeto, não pode ser um número; 3. A partir do segundo caractere podemos ter letra maiúscula, minúscula e números.
   1. [a-zA-Z][a-zA-Z0-9]{0,9}
6. Defina um *regex* para os links dos sites, que comecem com “file” e terminem com “html”. Ex: <file:///Users/nico/Downloads/regex/index.html>
   1. ^file.+html$
7. No sistema “Caused by: com.mysql.jdbc.exception.jdbc4.CommunicationsException: Communications link failure” faça uma regex garantindo que sempre irá começar com “Cause by:”
   1. ^Caused by:.\*
8. Para as datas “20 de maio de 2015” ou “20 maio 2015”, informe que não é necessário o uso do “de” para informar uma data (mas que se for inserido o “de” também não tem problema).
   1. (?:de\s+)?
9. Crie uma *regex* extraindo o nome do usuário contido no email. OBS: O email deve ter um “@” e terminar com “caelum.com.br” OU “alura.com.br”. O nome do usuário só possui letras minúsculas, pode ter um número no final e tem de 5 a 15 caracteres. Ex: super.mario@caelum.com.br, [donkey.kong@alura.com.br](mailto:donkey.kong@alura.com.br), [bowser1@alura.com.br](mailto:bowser1@alura.com.br)
   1. ([a-z.]{4,14}[a-z\d])@(?:alura.com.br|caelum.com.br)
10. Validando e-mails:
    1. ^(\w-]\.?)[+@([w-]+\.)+([A-Za-z]{2,4})+$](mailto:+@([w-]+\.)+(%5bA-Za-z%5d%7b2,4%7d)+$)
11. Agrupe o conteúdo da tag <h1 class=”text-left”>Expressões regulares</h1> OBS: Utilize *backreferences*
    1. <(h1|h2).+?>([w\sõãí.]+)<\/\1>

Utilizando regex no JavaScript:

var alvo = '11a22b33c';

// Coloca-se o "g" para informar que queremos todas as ocorrências encontradas da nossa expressão, não apenas a primeira que encontrar. (g = global)

var regex = /(\d\d)(\w)/g;

// uma outra forma de declarar: var regex = new RegExp(‘([\\d\\d)(\\w)](file:///\\d\\d)(\\w))’, ‘g’);

var resultado = null;

// Utilizamos regex.exec (para executar a regex)

while (resultado = regex.exec(alvo)){

    console.log (resultado);

    console.log (regex.lastIndex);

}

/\*

[ '11a', '11', 'a', index: 0, input: '11a22b33c', groups: undefined ]

3

[ '22b', '22', 'b', index: 3, input: '11a22b33c', groups: undefined ]

6

[ '33c', '33', 'c', index: 6, input: '11a22b33c', groups: undefined ]

9

\*/

Exercício: Trocando o hífem pela “/” em uma data.

var anoMesDia = '31/12/2022';

var exp = /-/g;

// Se você omitir o "g", seria trocado apenas o primeiro hífen.

anoMesDia = anoMesDia.replace(exp, '/');

// Aplicamos o método "replace" para fazer a substituição dos valores desejados. Ele recebe uma expressão regular como primeiro parâmetro e no segundo parâmetro, o novo texto que deve substituir todos os lugares que condizem com a expressão passada.

console.log(anoMesDia);

// 31/12/2022

Exercício: Um cliente quer separar os valores por vírgulas, porém os valores vieram misturados entre hífen, barras, ponto-vírgula...

* **Split( );** esse método divide uma *string* em uma lista ordenada de *substrings*, coloca essas *substrings* em um *array* e retorna o *array*. A divisão é feita procurando um padrão, onde o padrão é fornecido como o primeiro parâmetro na chamada do método.

var arquivo = '110,200-150,200;15';

var expressao = /[,;-]/;

var valores = arquivo.split(expressao);

// O método split cria um array com cada item considerando o separador informado. Nesse caso ele irá separar os itens que estão entre "," ";" e "-".

console.log(valores);

// [ '110', '200', '150', '200', '15' ]

Exercício: A variável código guarda uma string com vários códigos concatenados. A lei de formação é a seguinte: todo código começa com uma letra seguida de 1 ou mais dígitos. Precisamos extrair essas strings num array, ou seja, uma lista de código para que o desenvolvedor possa verificar se cada um desses códigos existe no sistema.

var codigos = 'A121B12112C12212F12G01';

var regularExp = /[A-Za-z]\d+/g;

var codigosExtraidos = codigos.match(regularExp);

// O método match aceita uma expressão regular e retorna em um array todas as partes do alvo que atendem a expressão regular.

console.log(codigosExtraidos);

//[ 'A121', 'B12112', 'C12212', 'F12', 'G01' ]

\* Utilizando a *tag* *pattern*: Essa *tag* recebe como atributo, expressões regulares, inclusive, quando o formulário for submetido e o valor digitado pelo usuário não for compatível com a expressão, o próprio navegador exibirá automaticamente uma mensagem para o usuário indicando que o campo é inválido. Mas é importante que o *input* faça parte de um formulário, caso contrário a validação não será aplicada.

<!DOCTYPE html>

<head>

    <meta charset="UTF-8">

    <title>Testando pattern</title>

</head>

<body>

    <form>

        <input pattern="[0-9]\*">

        <input type="submit" value="Enviar dados">

    </form>

</body>

</html>

**addEventListener** permite adicionar eventos a um determinado elemento.

* **removeEventListener** permite remover eventos a um determinado elemento.

Sintaxe: elemento.addEventListener(“evento”, função);

**preventDefault** cancela o evento se for cancelável, sem parar a propagação do mesmo. Ex: fazer com que ao clicar em um botão e o mesmo recarregar a página, isso não aconteça mais.

Sintaxe: event.preventDefault( );

**setTimeOut( );** é um método que determina um certo tempo (informado por nós) para então executar a função passada. OBS: o intervalo de tempo passado é definido em milissegundos (1segundo = 1000milsegundos)

sintaxe: let timeoutID = setTimeOut (function, intervalo\_de\_tempo);

**XMLHttpRequest( );** é o objeto responsável por fazer requisições HTTP assíncronas com o JS. El fornece funcionalidade ao cliente para transferir dados entre um cliente e um servidor. Oferece uma maneira fácil de recuperar dados de um URL sem ter que fazer uma atualização de página inteira. Isso permite que uma página web atualize apenas uma parte do conteúdo sem interromper o que o usuário esteja fazendo.

Sintaxe: let variável = new XMLHttpRequest( );

let botaoAdicionado = document.querySelector('#buscar-paciente');

botaoAdicionado.addEventListener('click', function(){

    let xhr = new XMLHttpRequest();

    xhr.open('GET', 'https://api-pacientes.herokuapp.com/pacientes');

    xhr.addEventListener('load', function(){

        console.log(xhr.responseText);

    });

    xhr.send();

})

**JSON** é a sigla para *JavaScript Object Notation*

**DOM** (Document Object Model) é a representação do HTML pro JavaScript. Para termos acesso ao *DOM* no JS, utilizamos o termo *document*

* **document.querySelector( );** para acessar algo dentro do navegador.
* **.textContent** para acessar o conteúdo de texto de dentro da tag HTML.

**toFixed(N);** para inserir um determinado número N de casas decimais ao valor.

* **document.createElement( );** cria um elemento no documento HTML
  + Sintaxe: let variavel = document.creatElement(‘nomeDaTag’);
* **.appendChild( );** insere um elemento dentro de outro (torna um elemento, filho de um elemento pai).
  + Sintaxe: elementoPai.appendChild(elementoFilho);

**InnerHTML** permite controlar e alterar o conteúdo interno do elemento desejado.

Sintaxe: elemento.innerHTML = “conteudo”.

**hasChildNodes( )** esse método retorna um valor *boolean* indicando se há um elemento filho dentro do elemento que foi passado. Não possui parâmetros.

**removeChild( )** remove um elemento filho do elemento que foi passado.

Sintaxes: var filhoRemovido = elemento.removeChild(filho);

elemento.removeChild(filho);

**.target** é uma referência ao objeto que enviou o evento.

Sintaxe: let variável = event.target

**.parentNode** retorna o elemento pai, do elemento filho referenciado.

Exemplo: Em uma tabela, ao dar um duplo clique numa célula, sua linha será removida do *table*. O *target* fará com que seja selecionado o elemento clicado (no caso a célula) e o *parentNode* faz com que suba para o elemento pai (no caso a linha da célula). OBS: Código do table está informado em um outro arquivo HTML.

let tabela = document.querySelector('table');

tabela.addEventListener('dblclick', function(evento){

let alvoEvento = evento.target;

let paiDoAlvo = alvoEvento.parentNode;

paiDoAlvo.remove();

});

**join( );** esse método junta todos os elementos de uma *array* em uma *string* e retorna essa *string.*

Sintaxe: *array*.join( );

OBS: Repare, como no exemplo abaixo, que se o separador for omitido, os elementos do *array* são separados com uma vírgula. Se o separador for uma *string* vazia, todos os elementos são juntados sem nenhum caractere entre eles.

const elementos = ['banana', 'melao', 'melancia'];

console.log(elementos.join());

// banana,melao,melancia

console.log(elementos.join(''));

// bananamelaomelancia

console.log(elementos.join('-'));

// banana-melao-melancia

**reverse( );** inverte os itens de um *array*. O primeiro elemento do *array* se torna o último e vice-versa. OBS: Não recebe parâmetros

Sintaxe: *array*.reverse( );

Exemplo: Transforme a data ‘’17-05-2016” para uma data no modelo “ano/mês/dia”

let dataString = '17-05-2016';

let data = new Date(dataString.split('-').reverse().join('/'));

console.log(data);

// 2016-05-17T03:00:00.000Z

**Static** define um método estático para a classe. Métodos estáticos não são chamados na instância da classe. Em vez disso, são chamados na própria classe. Geralmente são funções utilitárias, como funções para criar ou clonar objetos. Em resumo, o *static* é inserido antes do método da classe, quando esta mesma classe não possuirá instância, ou seja, será chamada diretamente a classe. Não pode ser feita em uma instância de classe.

Exemplo: O caso abaixo mostra como implementar o método *static* em uma *class* e como uma *class* com um membro *static* pode virar uma *subclasse*. Por último, ele mostra como o método *static* pode e não pode ser chamado.

class Triplo {

    static triplo (n) {

        n = n | 1;

        return n \* 3;

    }

}

class TriploMaior extends Triplo {

    static triplo (n) {

        return super.triplo(n) \* super.triplo(n);

    }

}

console.log(Triplo.triplo()); // 3

console.log(Triplo.triplo(6)); // 21

console.log(TriploMaior.triplo(3)); // 81

// O caso abaixo não é possível pois o método static não pode ser chamado por uma instância, apenas diretamente pela sua própria class.

var tp = new Triplo();

console.log(tp.triplo()); // Error

Exemplo: Chamando o método *static* dentro de outro método *static* da mesma classe. (Utilizamos o *this).*

class ChamadaMetodoEstatico {

    static metodoEstatico(){

        return 'O método estático foi chamado';

    }

    static outroMetodoEstatico(){

        return this.metodoEstatico() + 'de outro méotodo estático';

    }

}

console.log(ChamadaMetodoEstatico.metodoEstatico()); // O método estático foi chamado

console.log(ChamadaMetodoEstatico.outroMetodoEstatico()); // O método estático foi chamado de outro método estático

**defer** (adiar) diz ao navegador para executar o script apenas quando a análise do HTML estiver finalizada. O script será requisitado assincronamente, seu download será completado e apenas quando a análise do documento HTML estiver finalizada, ele será executado. Esse atributo é utilizado quando o script precisa que o DOM esteja totalmente carregado, e ao invés do *<script>* ser inserido ao final da tag *<body>,* ele permite que seja inserido em qualquer lugar da documentação, inclusive dentro da tag *<head>*.

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Exercicio-Defer</title>

    <link rel="stylesheet" href="style.css">

    <script src="script.js" defer></script>

</head>

<body>

</body>

</html>

**Switch** é uma condicional que avalia uma expressão, e está combinada a uma cláusula *case*, executando as instruções associadas ao *case*.

let permissao; //comum, gerente, diretor

switch (permissao) {

    case 'comum':

        console.log('usuário comum');

        break;

    case 'gerente':

            console.log('usuário gerente');

            break;

    case 'diretor':

            console.log('usuário diretor');

            break;

            default:

                console.log('usuário não reconhecido!');

}

**document.readyState** retorna “*loading”* enquanto o *document* está carregando, “*interactive”* quando foi carregado porém seus sub-recursos (como imagens, folhas de estilo, quadros) ainda não foram carregadas, e “*complete”* quando foi totalmente carregado.

* O evento **readystatechange**é adicionado no objeto *document* quando esse valor é alterado.

**DOMContentLoaded** esse evento é acionado quando todo o HTML foi completamente carregado e analisado, sem aguardar pelo CSS, imagens e subframes para encerrar o carregamento. Esse evento é muito diferente do *load*.

**Promises** são como promessas da vida real, ela está esperando algum retorno. Para criar uma promessa, instanciamos a class *Promise*. Ela leva dois argumentos: *resolve* (solução) e *reject* (erro). Para encadear mais processos utilizamos o método *then*. Alguns recursos de JS (Fetch API) e bibliotecas retornam *Promises*.