

## SUMÁRIO

<b>//JAVASCRIPT BÁSICO .....</b>	<b>2</b>
//* AULA4 DO COMANDO CONSOLE .....	2
//* AULA 8 - COMENTÁRIOS .....	2
//* AULA 09 - NAVEGADOR VS NODE (HTML+JAVASCRIPT).....	2
//* AULA 10 - VARIÁVEIS LET E VAR .....	3
//* AULA 11 - CONST .....	5
//* AULA 15 - PRIMEIRA DIFERENÇA ENTRE VAR E LET .....	6
//* AULA 16 - TIPOS DE DADOS PRIMITIVOS.....	6
//* AULA 17 - OPERADORES ARITMÉTICOS, DE ATRIBUIÇÃO E INCREMENTO. ....	7
//* AULA 18 - ALERT, CONFIRM E PROMPT (NAVEGADOR).....	9
//* AULA 21 - MAIS SOBRE STRINGS .....	10
//* AULA 23 - MAIS SOBRE NÚMEROS .....	12
//* AULA 24 - OBJETO MATH .....	14
//* AULA 26 - ARRAYS (BÁSICO).....	15
//* AULA 28 - FUNÇÕES (BÁSICO).....	17
//* AULA 29 - OBJETOS (BÁSICO).....	19
//* AULA 30 - VALORES PRIMITIVOS E POR REFERÊNCIA .....	21
<b>//LÓGICA DE PROGRAMAÇÃO .....</b>	<b>22</b>
//* AULA 33 - OPERADORES DE COMPARAÇÃO .....	22
//* AULA 34 - OPERADORES LÓGICOS.....	23
//* AULA 35 - AVALIAÇÃO DE CURTO-CIRCUITO (SHORT-CIRCUIT) .....	24
//* AULA 36 - IF, ELSE IF E ELSE (1).....	25
//* AULA 37 - IF, ELSE IF E ELSE (2).....	26
//* AULA 41 - OPERAÇÃO TERNÁRIA.....	26
//* AULA 42 - OBJETO DATE.....	27
//* AULA 43 - SWITCH/CASE .....	28
//* AULA 45 - MAIS DIFERENÇAS ENTRE VAR, LET E CONST .....	31
//* AULA 46 - ATRIBUIÇÃO VIA DESESTRUTURAÇÃO (ARRAYS).....	31
//* AULA 47 - ATRIBUIÇÃO VIA DESESTRUTURAÇÃO (OBJETOS).....	32
//* AULA 48 - FOR CLÁSSICO - ESTRUTURA DE REPETIÇÃO.....	33
//* AULA 50 - DOM E A ÁRVORE DO DOM.....	34
//* AULA 51 - FOR IN - ESTRUTURA DE REPETIÇÃO .....	34
//* AULA 52 - FOR OF - ESTRUTURA DE REPETIÇÃO.....	35
//* AULA 54 - WHILE E DO WHILE - ESTRUTURAS DE REPETIÇÃO.....	35
//* AULA 55 - BREAK E CONTINUE .....	37
//EXERCÍCIOS .....	38
//* AULA 59 - TRATANDO E LANÇANDO ERROS (TRY, CATCH E THROW).....	39
//* AULA 60 - TRATANDO E LANÇANDO ERROS (TRY, CATCH E FINALLY) .....	40
//* AULA 61 - SETINTERVAL E SETTIMEOUT.....	41
<b>//FUNÇÕES AVANÇADAS .....</b>	<b>42</b>
//* AULA 64 - MANEIRAS DE DECLARAR FUNÇÕES.....	42
//* AULA 65 - PARÂMETROS DA FUNÇÃO .....	43
//* AULA 66 - RETORNO DE UMA FUNÇÃO .....	44
//* AULA 67 - ESCOPO LÉXICO.....	45
//* AULA 68 - CLOSURES.....	46
//* AULA 69 - FUNÇÕES DE CALLBACK .....	46
//* AULA 70 - FUNÇÕES IMEDIATAS (IIFE) .....	48
//* AULA 71 - FUNÇÕES FÁBRICA (FACTORY FUNCTIONS).....	48
//* AULA 73 - FUNÇÕES CONSTRUTORAS (CONSTRUCTOR FUNCTIONS).....	49
//* AULA 75 - FUNÇÕES RECURSIVAS.....	50
//* AULA 76 - FUNÇÕES GERADORAS.....	51
<b>//ARRAYS AVANÇADO .....</b>	<b>53</b>
//* AULA 077 - REVISÃO DO BÁSICO DE ARRAYS .....	53
//* AULA 078 - MÉTODO SPLICE .....	54

/* AULA 079 - CONCATENANDO ARRAYS.....	55
/* AULA 080 - FILTER - FILTRANDO O ARRAY .....	55
/* AULA 081 - MAP - MAPEANDO O ARRAY .....	57
/* AULA 082 - REDUCE - REDUZINDO UM ARRAY .....	58
/* AULA 083 - FILTER + MAP + REDUCE .....	60
/* AULA 084 - FOREACH .....	61
<b>//OBJETOS PROTOTYPES .....</b>	<b>61</b>
/* AULA 85 - REVISANDO OBJETOS / MENÇÃO DO FREEZE .....	61
/* AULA 86 - OBJECT.DEFINEPROPERTY() E OBJECT.DEFINEPROPERTIES() .....	63
/* AULA 87 - GETTERS E SETTERS .....	64
/* <b>AULA</b> 88 - MÉTODOS ÚTEIS PARA OBJETOS.....	65
/* AULA 89 - PROPTOTYPES.....	66
/* AULA 90 - MANIPULANDO PROPTOTYPES.....	67
/* AULA 91 - HERANÇA .....	69
/* AULA 94 - RESOLUÇÃO VALIDA CPF .....	72
/* AULA 95 - POLIMORFISMO .....	73
/* AULA 96 - FACTORY FUNCTION + PROTOTYPES. ....	75
/* AULA 97 - OBJETO MAP().....	77
<b>//CLASSES .....</b>	<b>78</b>
/* AULA 98 - CRIANDO CLASSES.....	78

```
//JavaScript Básico
/* Aula4 do comando console
//#obs ele cita como aula 3 no vídeo, na udey está 4

//| diferenças ao apresentar uma string (texto)
//#nota a aspas duplas pode conter aspas simples e o aspas simples pode
envolver aspas duplas
//#nota o com crase pode conter ambas as aspas e é usado para template str
ing

console.log('Jean "Meira"');
console.log("Jean 'Meira'");
console.log(`Jean Meira`);

//| números
//#nota todos são tipo number, não muda de inteiro(int) para ponto flutuan
te(float)

console.log(15.85);
console.log(35);

//| Múltiplos valores

console.log(35, 15.85, 'Jean Meira de novo');

/* Aula 8 - comentários
//#nota# comentários são ignorados ao se executar (pela engine/motor)

// Isto é um comentário
console.log('Hello world'); // aqui temos outro comentário
//console.log('código não executado')

//#nota# códigos com (//) serão considerados comentários e não são executa
dos
/* isto é
   é um comentário
   que permite mais de uma linha,
   formando um comentário por bloco
   #importante# não deve se esquecer de fechar
*/

/* Aula 09 - navegador vs Node (HTML+JavaScript)

console.log('Este trecho será exibido no console do navegador, usando um a
rquivo .js')
```

```

//#nota# é possível criar uma pasta com todos scrips .js que serão usados
pelo arquivo
//#nota# assim é possível devinir um lugar com blocos convenientes de arqu
ivos .js

/* HTML A PARTIR DAQUI
<!DOCTYPE html>
<html lang="pt-br">
<head>
    <meta charset="UTF-8">
    <title>Minha primeira página HTML</title>

    <script>
        // um comentário
        console.log('olá mundo');
    </script>
</head>

<body>

    <script src='index.js'>
        // console.log('Este trecho será exibido no console do navegador')

/* código JS pode ser colocada dentro da tag scrip, mas é melhor
prática separar os arquivos, usando assim o apontamento do caminho (src)
para o arquivo .js, da forma a seguir
    <script src='index.js'>
*/
        // Um melhor prática é adicionar os scripts no final do body
    </script>
</body>
</html>

/* Aula 10 - variáveis let e var
//#obs# aula 6 citada no vídeo

//#nota# variáveis podem ser atribuídas como let e var

```

```

//#nota# podendo ou não ser inicializadas com valor, se não receber valor
será considerado undefined
//#obs# let se como variável do tipo let recebe o valor do tipo string
'João'.

let nome = 'João';           //? já inicializado na declaração

let nome2;                   //? declaração
nome2 = 'qualquer valor';    //? atribuição de um valor

console.log(nome, 'nasceu em 1984');
console.log('Em 2000', nome, 'conheceu Maria');
console.log(nome, 'caso-se com Maria em 2012');
console.log('Maria teve 1 filho com', nome, 'em 2015');
console.log('O filho de', nome, 'se chama Eduardo');

//#nota# var é mais antigo, o var permite redeclaração enquanto o let não,
ambos pode serem reatribuídos
//#nota# recebendo assim outros valores
//#aviso#let não podem ser declaradas mais de uma vez enquanto var pode
var nome3 = 'Jean';
var nome3 = 'Guilherme';

console.log(nome3);

//#obs# A variável permite salvar valores que serão usados em múltiplas pa
rtes do código, além de trabalhar
//#obs# informações no decorrer do código e poder salvar informações vinda
s do usuário ou fontes externas.

//#importante# o uso de variáveis ou partes dinâmicas de código podem mais
facilmente ser alteradas,
//#importante# o código exemplificado acima pode mudar o nome citados em t
odas as frases somente alterando
//#importante# a linha de declaração da variável nome, o que não seria pos
sível se fosse manualmente
//#importante# escrito em cada comando console.log();

//#nota# Não podemos criar variáveis com palavras reservadas ex -> let if
-- let let, e assim por diante;
//#nota# É recomendado que variáveis tenham nomes significativos ex -> let
n = 'João';
//#nota# n é muito vago, podendo ser qualquer coisa
//#obs# É bom que a variável tenha valor semântico.

```

```

//#nota# Não começar o nome de uma variável com um número;
//#nota# Variáveis em geral começam com letras minúsculas, (existem exceções);
//#nota# Não podem conter espaços ou -, ex -> let nome cliente; let nome-completo;
    //#obs# para variáveis com nomes múltiplos pode-se usar o padrão camelCase ex -> let nomeCLiente.
//#nota# Variáveis são case-sensitive, ou seja, nas variáveis letras maiúsculas e minúsculas diferem
//#nota# no resultado final;

//#importante# NÃO UTILIZE VAR, UTILIZE LET

//* Aula 11 - Const

//* As mesmas regras de variáveis valem para constantes
//#nota# Não podemos criar constantes com palavras reservadas ex -> let if -- let let, e assim por diante;
//#nota# É recomendado que constantes tenham nomes significativos ex -> let n = 'João';
//#nota# n é muito vago, podendo ser qualquer coisa
    //#obs# É bom que a constante tenha valor semântico.
//#nota# Não começar o nome de uma constante com um número;
//#nota# constantes em geral começam com letras minúsculas, (existem exceções);
//#nota# Não podem conter espaços ou -, ex -> let nome cliente; let nome-completo;
    //#obs# para constantes com nomes múltiplos pode-se usar o padrão camelCase ex -> let nomeCLiente.
//#nota# constantes são case-sensitive, ou seja, nas variáveis letras maiúsculas e minúsculas diferem
//#nota# no resultado final;
//#importante# Constantes não podem ser redeclaradas e nem tem novos valores atribuídos as mesmas.
    //#obs# não podem modificar seu valor
//#importante# devem ser inicializadas e declaradas ao mesmo tempo

const nome = 'João';

console.log(nome);

//#nota# É possível usar uma constante ou variável na declaração de outra.

const primeiroNumero = 5;
const primeiroNumeroStrg = '5';

```

```

const segundoNumero = 10;
const resultado = primeiroNumero * segundoNumero;
const resultadoDuplicado = 2* resultado;

console.log(resultado); //? resposta es
perada -> 50
console.log(resultadoDuplicado); //? resposta es
perada -> 100

/* Verificando o tipo de uma const ou let com a função typeof();

console.log(primeiroNumero + segundoNumero); //? resposta e
sperada -> 15
console.log(typeof(primeiroNumero + segundoNumero)); //? resposta e
sperada -> number
console.log(primeiroNumeroStrg + segundoNumero); //? resposta e
sperada -> 510
console.log(typeof(primeiroNumeroStrg + segundoNumero)); //? resposta e
sperada -> string

//#nota# Ao receber uma string e um number o código concatena os valores e
m vez de somar,
//#nota# apresentando a string 5 e o número 10 escritos em sequência.
//#nota# Porém, dessa forma o resultado passa a ser interpretado como uma
string.

/* Aula 15 - primeira diferença entre var e let
//#obs# no vídeo citado como aula 9

//? Var Aceita redeclaração

var nome = 'Luiz'; //? Declaração
var nome = 'Otávio' //? Re-declaração

console.log(nome); //? resultado esperado -> Otávio

//#aviso# Adendum não faça o demonstrado a seguir
nome1='Luiz';

/* Aula 16 - Tipos de dados primitivos

//| strings

const nome = 'Luiz';
const nome1 ="Luiz";

```

```

const nome2 = `Luiz`;

/// numbers

const num1 = 10;
const num2 = 10.5;

/// undefined
let nomeAluno           //? undefined -> não aponta para nenhum local
na memória

/// null
let sobernomeAluno = null     //? null -> não aponta para nenhum local na me
mória

///#nota# undefined != de null
///#nota# null é a indicação que foi escolhido que a variável não terá um v
alor, não aponta para nenhuma memória
///#nota# undefined é uma variável que não recebeu um valor.

///? boolean
const boolean = true;
const boolean2 = false;

///#nota# boolean assume valor true ou false, representando verdadeiro ou f
also, 1 ou 0;
///#obs# boolean tem peso lógico, ajuda em decisões do código. Muda o fluxo
da aplicação, usando desvios condicionais.

/* Aula 17 - Operadores aritméticos, de atribuição e incremento.

/// Aritméticos
///_ adição          -> +
///_ subtração       -> -
///_ divisão         -> /
///_ multiplicação   -> *
///_ potenciação     -> **
///_ resto da divisão-> %

///#nota# a precedência das operações são realizadas conforme a matemática

///_ exemplo
const num1 =5;
const num2 =2;
const num3 =10;

```



```

console.log(num1 + num2 * num3);    //? resultado esperado -> 25

/*
    ! Ordem de precedência segue conforme indicado abaixo
    _ 1º ** (potenciação)
    _ 2º * (multiplicação) , / (divisão) e % (resto da divisão ou módulo d
a divisão)
    _ 3º + (adição) e - (subtração)
*/

//#obs# é possível alterar a ordem de precedência com o uso parênteses ()
//_ exemplo

console.log((num1 + num2) * num3);    //? resultado esperado -> 70

//| Incremento e decremento
//_ ++ soma um no valor (incrementa)
//_ -- subtrai um no valor (decrementa)

//_ exemplo (funcionam para incremento e decremento)

let contador =1;
contador++;
console.log(contador);

console.log(contador++)                //? realiza a ação e depois incrementa
    (pós incremento)
    //#obs# evite usar o modelo acima, com pós incremento junto com a func
ão que irá usar de imediato, pois pode causar bugs.

console.log(++contador)                //? incrementa e depois realiza a ação
    (pré incremento)

//| Operadores de atribuição
//_ incremento de mais de uma unidade

const passo =2;
let contador1 =0;

contador1 = contador1 + passo;
console.log(contador1);

//_ De modo simplificado

```

```

//#obs# é possível usar com incremento, decremento, multiplicação, divisão
e potenciação

contador1 += passo;           //? o mesmo que digitar -> contador1 = contado
r1 + passo;
console.log(contador1);

/*
    ! cuidado com usar contas ou atribuições com variáveis, ao se usar com
    variáveis com tipagem diferentes de números
    ! pode se obter resultados adversos e inesperados.
    ! podendo até obter resultados NaN -> not a number
*/

//#importante# Quando for possível converta as variáveis para o tipo deseja
do para garantir o funcionamento

//_ Exemplo
const numTest = parseInt('5');

console.log(typeof(numTest));           //? resultado esperado -> number

//_ parseInt()      -> converte para inteiro, sem números após a vírgula
//_ partseFloat()   -> converte para float, números com valores após a vír
gula
//_ Nuber()         -> converte para número, sem distinção

//* Aula 18 - Alert, confirm e Prompt (Navegador)

//| Alert
alert('Mensagem');

//#nota# alert é um método do objeto window
//#obs# alert é um atalho para window.alert();
//_ o retorno é undefined, ou seja, não retorna valor algum.

//| Confirm
window.confirm('Deseja realmente apagar?');

//| prompt
//? sempre vai te retornar uma string
window.prompt('Digite o seu nome.');
```

//| é possível capturar o retorno da função

```

const confirma = confirm('Realmente deseja apagar?');
console.log('confirma tem valor:', confirma);

let num1 = prompt('Digite um número');
alert(`Você digitou: ${num1}`);
console.log(`Você digitou: ${num1}`);

//| A partir daqui é HTML
<!DOCTYPE html>
<html lang="pt-BR">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Aula 18</title>
</head>
<body>
  <script src="./script.js"></script>
</body>
</html>

/* Aula 21 - Mais sobre Strings

let umaString = "Um \"texto\""; //? a barra invertida é um caractere de escape.
//_ dessa forma é possível inserir aspas duplas em uma string declarada com aspas duplas

console.log(umaString);

umaString = "Um \\texto"; //? usando duas barras é possível fazer uma barra aparecer no resultado final

console.log(umaString);

//#nota# uma string é indexável. Cada caractere tem um índice. Começando em 0 (zero);
umaString = "Um texto";
console.log("A quinta letra da variável umaString é: "+umaString[4]);
console.log(umaString.charAt(4)); //? usando o método charAt(); para realizar o mesmo.
console.log(umaString.charCodeAt(4)); //? retorna o código da tabela ASCII.

//| função concat concatena os textos, como o + ou uso de templateString

```

```

console.log(umaString.concat(' em', ' um', ' lindo dia'));
console.log(umaString + ' em' + ' um' + ' lindo dia');
console.log(`${umaString} em um lindo dia`);

//| Pesquisando índice que começa uma palavra
console.log(umaString.indexOf('texto'));
//#nota# retorna o índice se achar e -1 se não achar.

//#obs# é possível enviar dois parâmetros, então o segundo será de onde a
pesquisa começa
console.log(umaString.indexOf('um', 3));    //? retorno -
1 pois não há um após o índice 3.

console.log(umaString.lastIndexOf('o'));    //? começa a procura do final
para o começo.
//#obs# pode dar diferença se mandar o parâmetros de início.

console.log(umaString.search(/x/));          //? similar ao indexOf, mas ac
eita expressões regulares.
//#obs# e em realidade pode ter funcionamentos um pouco mais amplos.

//| Replace
console.log(umaString.replace('Um', 'outra')); //? substitui uma palavra p
or outra

umaString = 'O rato roeu a roupa do rei de roma';
console.log(umaString.replace(/r/, '#'));    //? substitui somente o 1º r
console.log(umaString.replace(/r/g, '#'));    //? substitui todos os r

//| Checando o tamanho

console.log(umaString.length);
//          012345
umaString = 'O rato';
console.log(umaString.length);    //? conta como 0 a 5, ou seja 6

//| dividindo uma string

umaString = 'O rato roeu a roupa do rei de roma.';

console.log(umaString.slice(2, 6));
//#nota# Indica a posição de início e final, sendo que o final não é
contado.

console.log(umaString.slice(2));

```

```

//#nota# Se não receber o segundo parametro, conta como do início indicado
até o final.

console.log(umaString.length-3);           //? resultado esperado -> 32
console.log(umaString.slice(-3));
console.log(umaString.slice(32));
/*
    _ Usar número negativo é o mesmo que adotar o início como o tamanho to
tal do texto
    _ menos o valor indicado. Sendo assim, no exemplo acima é printado ape
nas os 3
    _ últimos caracteres. E -3 é o mesmo que indicar o início de 32.
    _ Também é possível indicar começo e final com números negativos.
*/

//? também é possível usando outro comando.
console.log(umaString.slice(-5, -1));
console.log(umaString.substring(umaString.length - 5, umaString.length -
1));

//? Baseado em algum caractere.

console.log(umaString.split(' '));
console.log(umaString.split('r'));

console.log(umaString.split(' ', 2));           //? é possível limitar o númer
o de resultados.

//| representar em maiúsculo ou minúsculo
console.log(umaString.toUpperCase());
console.log(umaString.toLocaleLowerCase());

//* Aula 23 - Mais sobre números

let num1 =1;    // numbe
let num2 =2.5;  // number

console.log(num1 + num2);
console.log(num1.toString()+ ' e ' + num2.toString());
console.log(typeof num1);

//_ É realizado a contenção, mas o tipo do número permanece inalterável.

//? para converter -> num1 = num1.toString();

```

```

num1 = 15;
console.log(num1.toString(2));  //? representa o número em base binária.

num1 = 10.872158137674;
console.log(num1.toFixed(2));  //? limita o número de casas após a vírgula
a que será mostrado
//#obs# o número é arredondado.
//#aviso# recomenda-se fazer todos os
cálculos e apenas usar esse artifício ao exibir o resultado.

console.log(Number.isInteger(num1));  //? o número é um tipo inteiro ?
//_ Retorna verdadeiro ou falso.

let temp = num1 * 'olá';
console.log(temp, Number.isNaN(temp));  //? a conta é inválida, retornando
um NaN;

//! padrão IEEE 754-2008 -> para imprecisão

num1 = 0.7;
num2 = 0.1;

console.log(num1+num2);  //? resultado -> 0.79999...

//? o mesmo que num1 + num2;
num1 += num2;  //? 0.8
num1 += num2;  //? 0.9
num1 += num2;  //? 1.0

console.log(num1);  //? mas dá o resultado 0.9999...

num1 = num1.toFixed(2);  //? fixa apenas duas casas;
console.log(num1);  //? o resultado dá 1.00, parecendo correto.
console.log(Number.isInteger(num1), Number.isInteger(1.00));  //? aqui nota-se a diferença.

//| Para corrigir o problema devemos...
num1 = 0.7;
num2 = 0.1;

num1 += num2;  //? 0.8
num1 += num2;  //? 0.9
num1 += num2;  //? 1.0

```

```

num1 = Number(num1.toFixed(2));    //? fixar as casas garantindo que será
    um número

console.log(num1);
console.log(Number.isInteger(num1));    //? só será inteiro se as casas ap
ós o ponto forem .00

//| outra forma seria trabalhar com contas.
//_ não somando números com vírgulas
num1 = 0.7;
num2 =0.1;

num1 = ((num1*100)+(num2*100))/100;
console.log(num1);    //? 0.8

num1 = ((num1*100)+(num2*100))/100;
num1 = ((num1*100)+(num2*100))/100;

console.log(num1);
console.log(Number.isInteger(num1));

/* Aula 24 - Objeto Math

/*
    _ Math é um objeto embutido que tem propriedades
    _ e métodos para constantes e funções matemáticas.
    _ Não é um objeto de função.
    _ Todas as propriedades e métodos de Math são estáticas.
*/

let num1 = 9.54578;

//| arredondamento
//_ Para baixo
let num2 = Math.floor(num1); //? arredondando para baixo.
console.log(num2);    //? resultado esperado -> 9;

//_ Para cima
num2 = Math.ceil(num1); //? arredonda pra cima.
console.log(num2);    //? resultado esperado -> 10;

//_ Para o que estiver mais próximo
num2 = Math.round(num1); //? arredonda para o mais próximo
console.log(num2);    //? resultado esperado -> 10

```

```

num1 = 9.44578;
num2 = Math.round(num1);
console.log(num2);  //? resultado esperado -> 9

num1 = 9.50;
num2 = Math.round(num1);
console.log(num2);  //? resultado esperado -> 10
//#obs# Se o número estiver na metade será considerado para cima

//| O maior e menor número de uma sequência
console.log(Math.max(1,2,3,4,5,-10,-
50,1500,9,8,7,6));  //? resultado esperado -> 1500
console.log(Math.min(1,2,3,4,5,-10,-
50,1500,9,8,7,6));  //? resultado esperado -> -50

//| gerando um número aleatório entre 0 e 1, sem incluir o 1;
console.log(Math.random());

//_ é possível trabalhar para gerar outros resultados
const aleatorio = Math.round(Math.random() * (10-5) + 5);
console.log(aleatorio);  //? aleatório entre 5 e 10.

//| o Math tem constantes
console.log(Math.PI);  //? o pi = 3.14158...
console.log(Math.E)    //? o número de Euler = 2,71828...

//| Potenciação
console.log(Math.pow(2, 10));  //? 2 elevado a 10
//#obs# pode-se dizer que é o mesmo que
console.log(2 ** 10);          //? 2 elevado a 10

//| Raiz
num1 = 9;
console.log(num1 ** (1/2));
console.log(Math.sqrt(num1));

//! Cuidado com divisões por 0, pois no Js isso é possível.
//? ex:
console.log(100 / 0);  //? o número retornará infinity

//* Aula 26 - Arrays (básico)

const alunos = ['Jean', 'Maria', 'João'];
console.log(alunos)
//_ tentar organizar com um tipo só de dados dentro, mas não é regra

```



```
//_ o JavaScript permite a adição, mas não é boa prática de programação

//_ Arrays são indexados por elementos, começando no valor 0.
console.log(alunos[0])

//| é possível editar ou adicionar
alunos[1] = 'Roberta';
alunos[3] = 'Luiza';
console.log(alunos);

// _ Se o Array tem um elemento na posição ele altera, se não ele adiciona
.

//| Se quiser adicionar no último é possível usar conforme a seguir.
console.log(alunos.length)
alunos[alunos.length] = 'Fábio';
alunos[alunos.length] = 'Luana';
alunos.push('Otávio');
console.log(alunos);

//| Se quiser adicionar no começo...

alunos.unshift('Luiza');
console.log(alunos);

//| Removendo
alunos.pop();    //? remove o último
//_ é possível salvar o elemento removido
const removido = alunos.pop();
console.log(alunos, removido);

alunos.shift(); //? remove do começo
console.log(alunos);

delete alunos[2];    //? remove o índice específico
console.log(alunos, alunos[2])
//_ O delete não muda os índices, o que ocorre com pop e shift. O local ap
agado fica como undefined para o caso do delete;

alunos[2] = 'Carlos';
//| Dividindo um Array
console.log(alunos.slice(0, 3));
console.log(alunos.slice(0, -1));

//_ Arrays retornar como objetos, pois são como objetos indexados
```

```
console.log(typeof alunos);
//? é possível checar se é um array
console.log(alunos instanceof Array);

//#Aviso# geralmente é melhor adicionar elementos no final do array, para
que não se tenha que mudar todas as posições.
//#Aviso# isso torna a performance do programa ruim.

/* Aula 28 - Funções (básico)

//#nota# funções executam ações, podendo ou não retornar algo

function saudacao(){
    console.log('Bom dia');
};

saudacao();

//| funções que recebem parâmetros

function saudacao1(nome){
    console.log(`Bom dia ${nome}`);
};

saudacao1('Jean');

//#nota# função são reutilizáveis
saudacao1('Fulano');

const variavel = saudacao1('Fulano');
console.log(variavel);
//#obs# toda função por padrão retorna undefined, e por isso é salvo unde-
fined na variavel.

//| Funções com retorno

function saudacao2(nome){
    console.log(`Bom dia ${nome}`);
    return 123456
};

const retorno = saudacao2('Jean');
console.log(retorno);
```

```

//#aviso# Esperasse que o retorno da função seja algo semantico com o nome
, e não que seja algo bem diferente.

function saudacao3(nome){
    return `Bom dia ${nome} pelo retorno`
};

const retorno2 = saudacao3('Jean');
console.log(retorno2);

//| funções com mais uso e que permitem reprimir o seu uso.

function soma(x,y){
    const resultado = x + y;
    return resultado;
}

console.log(soma(2,2));
console.log(soma(3,1));
console.log(soma(5,10));

// console.log(soma(2,2), resultado); //#nota# daria erro pois o resultado
faz parte do escopo local da função.

const resultado = soma(2,2);    //? é possível declarar uma const resultad
o, pois a que está dentro da função está isolada.

//! Quando a função chegar em um return sairá da função e aplica o retorno
, tudo após um return será ignorado.

function soma1(x,y){
    const resultado = x + y;
    return resultado;
    console.log('olá mundo');
}

console.log(soma1(5,10));

//| Valores faltando

console.log(soma()); //? retorna NaN
//#obs# será NaN mesmo se faltar só um

//_ É possível tratar atribuindo um valor inicial para os argumentos da fu
nção

```

```

function soma2(x=1,y=1){
    const resultado = x + y;
    return resultado;
}

console.log(soma2());
console.log(soma2(5,10));

//#obs# caso não receba valores, serão usados os pré-
definidos, caso receba será adotado o recebido.

//| Outros modos de criar funções

//_ Funções anônimas -> em variáveis.
//#obs# precisa do ponto e vírgula obrigatoriamente no final.

const raiz = function(n){
    return (n ** 0.5);
};

console.log(raiz(9));
console.log(raiz(16));
console.log(raiz(25));

//_ Arrow function

const raizArrow = (n) => {
    return ( n ** 0.5);
};

console.log('Arrow', raizArrow(9));

//_ Se tiver apenas uma linha de retorno é possível simplificar ainda mais
.

const raizArrow1 = (n) => n ** 0.5;
console.log('Arrow resumida',raizArrow1(9));

//#Aviso# funções são basicamente iguais, mesmo com as declarações diferen
tes.
//#Aviso# Quando entrar em this terão diferenças que serão expostas
//importante# Não é uma boa prática criar funções que executem diversas a
ções.

/* Aula 29 - Objetos (básico)

```

```

//#Nota# é possível alterar os valores dos elementos de objetos e arrays.
//#Nota# mas não é possível reatribuir o valor ou mudar o tipo.
//#Nota# ao mudar elementos internos não há alterações pra onde é apontado
na memória.

//| declaração

const pessoa1 = {
  nome: 'Luiz',
  sobrenome: 'Miranda',
  idade: 25
};

console.log(pessoa1.nome);
console.log(pessoa1.sobrenome);
//#Nota# da pra criar atributos que são como variáveis, mas estão internos
ao objeto.
//#Nota# os atributos são separados por uma , no final de cada linha
//#Nota# e num objeto utiliza par chave e valor para cada atributo, que são
separados por :

//| criar por funções -> function factory
// fábrica de objetos.

function criaPessoa(nome, sobrenome, idade){    //? Isso são parâmetros de
  uma função
  return{ nome, sobrenome, idade};
  /* É o mesmo que citar como nome: nome e assim por diante,
  Pois quando a chave e o valor possuem o mesmo nome é possível
  omitir o : valor
  */
}

const pessoa2 = criaPessoa('Jean','Meira','23');    //? Aqui são argumentos
que serão passados para os parâmetros.
console.log(pessoa2);

//#Nota# argumentos são os valores que são passados para o parâmetro.

//| Criar método no objeto

const pessoa3 = {
  nome: 'Jean',
  sobrenome: 'Meira',

```

```

    idade: 23,
    fala(){
        console.log(`${this.nome} ${this.sobrenome} está falando oi...
        A minha idade é ${this.idade}`)
    },
    incrementaIdade(){
        this.idade++;
    }
}

pessoa3.fala();
pessoa3.incrementaIdade();
pessoa3.fala();

//#Nota# this referencia o contexto da função, mas será dado mais detalhes
a frente.

/** Aula 30 - Valores primitivos e por referência

/*
_ Primitivos (imutáveis) - string, number, boolean, undefined, null
_ também existem os bigint e symbol
_ dado é o valor, a variável é somente uma caixa que contém o valor.
*/

let nome = 'Luiz';
nome = 'Otávio';
console.log(nome);
//#Obs# apesar de trocar o que está escrito, não se altera o dado / valor
primitivo

let a = 'A';
let b = a; // é feito uma cópia
console.log(a, b);

a = 'Outra coisa';
console.log(a, b);
//_ b não se altera por ser uma cópia de a, então alterando a o b não é af
etado.

/*
_ Por Referência (mutável) - array, object , function
! são passados por referência
_ O que na verdade quando são atribuídos a outras variáveis,
_ significa que vão apontar para a mesma referência, o mesmo local

```

```

    _ na memória.
*/

let c = [1, 2, 3];
let d=c;    // d vai apontar para o mesmo lugar na memória
console.log(c,d);

c.push(4);
console.log(c,d);

//_ b é afetado, pois o local na memória é o mesmo, então mudando por a ou
por b, afeta ambos
//? exemplo
d.pop();
console.log(c,d);

//| para cópiar o valor e mudar o local apontado na memória

let e = [1, 2, 3];
let f= [...e];
e.push(4);
console.log(e,f);

//Lógica de programação
//* Aula 33 - Operadores de comparação

/*
_ >      maior que
_ >=     maior que ou igual a
_ <      menor que
_ <=     menor que ou igual a
_ ==     igualdade (checa valor)
_ ===    igualdade estrita (checa valor e tipo)
_ !=     diferente (checa valor)
_ !==    diferente estrito (checa valor e tipo)

*/

console.log(10>5);
//? É possível salvar o valor em variáveis e constantes
const comp = 10 > 5;
console.log(comp);

//_ A funcionalidade é igual as usadas em matemática. Os estritos tem um c
onceito a mais.

```

```

//| Com variáveis

let num1 = 10;
let num2 = 11;

console.log(num1 <= num2);

//| Comparação e comparação estrita

num1 = 10;
num2 = 10;
console.log(num1 == num2);
num2 = '10';
console.log(num1 == num2);
console.log(num1 === num2);

/*
_ A comparação normal somente compara o valor contido, enquanto
_ a comparação estrita compara o valor e o tipo da variável ou
_ dado em questão. Isso é válido para a igualdade e desigualdade.
! O uso do modo estrito é altamente recomendado para evitar
! Comportamentos indesejados na execução do código.
! EVITE O USO DA IGUALDADE E DESIGUALDADE COMUNS
*/

/** Aula 34 - Operadores Lógicos

/*
_ Operadores lógicos
_ && -> and -> e
_ || -> or -> ou
_ ! -> not -> não
*/

//| && (and)
//_ Para ser verdadeiro todas as expressões precisam ser verdadeira
console.log(true && true);
console.log(true && true && true && true);

//_ É possível salvar o valor em uma variável ou constante
const expressaoAnd = (true && true && true && true);
console.log(expressaoAnd);

```



```
//_ Se uma for falsa o resultado já é falso
console.log(true && false);
console.log(true && true && false && true);

//| || (or)
//_ Para ser verdadeiro pelo menos uma das expressões precisam ser verdade
ira
console.log(true || true);
console.log(true || false);

//_ É possível salvar o valor em uma variável ou constante
const expressaoOr = (true || true);
console.log(expressaoOr);

//_ Se uma for falsa se todas as expressões forem falsas
console.log(false || false);
console.log(false || false || false || false);

//| ! (not)
```

```
//_ Sai o oposto do que entrou
console.log("negação de true:", !true);
console.log("negação de dupla de true:", !!true);

/** Aula 35 - Avaliação de curto-circuito (short-circuit)

/*
  _ && -> false && true -> false : retorna "o valor" ao achar uma falsa

  | FALSY (VALORES QUE PODEM SER AVALIADOS COMO FALÇO)
  _ false
  _ 0
  _ '' "" `` (string vazias)
  _ null / undefined
  _ NaN

  #nota# Qualquer valor diferente dos presentes no FALSY avalia verdadei
ro
*/

//? exemplo
/*
  _ Ao se deparar com um FALSY o valor do mesmo é retornado, se em uma
  _ comparação não tiver nenhum FALSY a mesma irá retornar o último valo
r lido.
```

```

    _ Isso possibilita fazer uma redução nos circuitos para alguns casos,
    tornando o
    _ código mais limpo e performático.
*/
console.log("Jean" && 0);    ////? retorna 0;

/*
    _ A seguir tem um caso de curto circuito, onde o código é omitido em p
    artes
    _ Porque devido as propriedades de falsy o código é perfeitamente váli
    do.
    _ Será testado a condição e ao perceber se vaiExecutar é verdadeiro ou
    falso
    _ já retornará o resultado.
*/
function falaOi (){
    return 'Oi';
}

const vaiExecutar = false;

console.log(vaiExecutar && vaiExecutar);

/*
    _ 0 || (or) tem comportamento ao contrário
    _ || -> false && true -> true : retorna "o valor" ao achar uma true

    | FALSY (VALORES QUE PODEM SER AVALIADOS COMO FALÇO)
    _ false
    _ 0
    _ '' "" `` (string vazias)
    _ null / undefined
    _ NaN

    #nota# Qualquer valor diferente dos presentes no FALSY avalia verdade
    iro
*/

/** Aula 36 - if, else if e else (1)

const hora = 12;

if (hora >= 0 && hora < 12){
    console.log('Bom dia');
} else if (hora >= 12 && hora < 18){

```

```

    console.log('Boa tarde');
} else if (hora >= 18 && hora <=23){
    console.log('Boa noite');
}else {
    console.log('Olá');
}

/*
_ If é o primeiro teste de condição, que ser verdadeiro, executa o código contido nas chaves.
_ if pode ser usado sozinho.
_ Se a condição for falsa pode ter outras condições diversas com o else if, em qualquer quantidade.
_ O else if precisa ver seguido de um if, não podendo ser usado sozinho.
_ Se todas as condições forem falsas será executado o conteúdo do else.
_ O else vem no final e só pode ser usado um.
*/

/** Aula 37 - if, else if e else (2)

//_ No uso de if e else, os blocos são interdependentes. De modo direto
//_ Se um ocorrer o outro não ocorre.

//_ No caso de usar if, else if (em quantidade desejada) e else
//_ O else depende do if e de todos else if, se não houver verdadeiros executa o else;

//_ O bloco de if, else if e else para quando encontra um elemento verdadeiro.

//_ Cada bloco com if é independente de outro if.
*/
/** Aula 41 - Operação ternária

//#nota# são o conjunto de ? :
//#nota# sendo (condição) ? (valor para verdadeiro) : (valor para falso)

const pontuacaoUsuario = 999;

if(pontuacaoUsuario >= 1000){
    console.log('Usuário vip');
}else{
    console.log('Usuário normal');
}

```

```

const nvUsuario = pontuacaoUsuario >= 1000 ? 'Usuário vip' : 'Usuário normal';
console.log(nvUsuario);

//#obs# funciona como um if e else, apenas um teste com retorno se verdadeiro ou falso.

/* Aula 42 - Objeto Date

//| formato em branco -> momento atual
const data = new Date();    //? é contada em milésimos de segundos
console.log(data.toString());

//| fomato valor
const data0 = new Date(0); //? 01/01/1970 Timestamp unix ou época unix
//#nota# esse é o marco zero da era unix, e para datas posteriores deve ser valores positivos
//#obs# e para anteriores negativos

console.log(data0.toString()); //? mas o resultado é:
    //? Wed Dec 31 1969 21:00:00 GMT-0300 (Horário Padrão de Brasília)
    //#obs# isso se deve ao fuso horário, se for somado 3 passar a ser a data 01/01/1970

//| formato ano,mes,dia,hora,minuto,segundo,milissegundo

const dataEscolhida = new Date(2019, 3, 20, 15, 14, 27, 500); //?ano, mes, dia, hora, min, seg, milésimo
//#nota# 0 mês se conta do 0 ao 11
//_ os milésimos vão até 999, se colocar 1000 ou mais ele passa 1 para a diante, corrigindo.
//_ assim como segundos de 0 a 59 e assim por diante.
//_ Se omitir algo ele aceitará como zero
//_ Não é possível omitir um valor e colocar o mais a direita dele.
//_ ex: se colocar o ano, omitir o mês e colocar o dia, os dias serão contados como meses.
//_ mas é possível omitir os mais a direita, até no máximo os meses, sendo necessário indicar o ano.
console.log(dataEscolhida.toString()) //? Sat Apr 20 2019 15:14:27 GMT-0300 (Horário Padrão de Brasília)

//| formato datastring

const dataString = new Date('2019-04-20 20:20:59.100');

```

```

const dataString1 = new Date('2019-04-20T10:10:59.599');
console.log(dataString.toString());
console.log(dataString1.toString());

//| obter dia
console.log('Dia', data.getDate()); //? -> dia de domingo (0) a sábado (6)
console.log('Mês', data.getMonth()+1);
console.log('Ano', data.getFullYear());
console.log('Hora', data.getHours());
console.log('Min', data.getMinutes());
console.log('Seg', data.getSeconds());
console.log('ms', data.getMilliseconds());
console.log('Dia semana', data.getDay());

//| obter os milésimos de segundo de agora sem new Date();
console.log(Date.now()); //? -> retorna em milésimos.

//| formatando data

function zeroAEsquerda(num){
    return num >= 10 ? num : `0${num}`
}

function formataData(data){
    const date = zeroAEsquerda(data.getDate());
    const month = zeroAEsquerda(data.getMonth()+1);
    const year = zeroAEsquerda(data.getFullYear());
    const hours = zeroAEsquerda(data.getHours());
    const minutes = zeroAEsquerda(data.getMinutes());
    const seconds = zeroAEsquerda(data.getSeconds());

    return `${date}/${month}/${year} ${hours}:${minutes}:${seconds}`
}

const dataBrasil = new Date();
const dataAgora = formataData(dataBrasil);
console.log(dataAgora);
//* Aula 43 - Switch/Case

//_ estrutura condicional switch case

const data = new Date('1987-04-21 00:00:00');
const diaSemana = data.getDay();
let diaSemanaTexto;

```

```
if (diaSemana === 0) {
    diaSemanaTexto = 'Domingo';
} else if (diaSemana === 1) {
    diaSemanaTexto = 'Segunda';
} else if (diaSemana === 2) {
    diaSemanaTexto = 'Terça';
} else if (diaSemana === 3) {
    diaSemanaTexto = 'Quarta';
} else if (diaSemana === 4) {
    diaSemanaTexto = 'Quinta';
} else if (diaSemana === 5) {
    diaSemanaTexto = 'Sexta';
} else if (diaSemana === 6) {
    diaSemanaTexto = 'Sábado';
} else {
    console.log('Erro');
}

console.log(diaSemanaTexto);

const dataSC = new Date('1987-04-22 00:00:00');
const diaSemanaSC = dataSC.getDay();
let diaSemanaTextoSC;

switch (diaSemanaSC) {
    case 0:
        diaSemanaTextoSC = 'Domingo';
        break;
    case 1:
        diaSemanaTextoSC = 'Segunda';
        break;
    case 2:
        diaSemanaTextoSC = 'Terça';
        break;
    case 3:
        diaSemanaTextoSC = 'Quarta';
        break;
    case 4:
        diaSemanaTextoSC = 'Quinta';
        break;
    case 5:
        diaSemanaTextoSC = 'Sexta';
        break;
    case 6:
        diaSemanaTextoSC = 'Sábado';
}
```

```

        break;
    default:
        console.log('Erro');
}

console.log(diaSemanaTextoSC);

///| dentro de função

function getDiaSemanaTexto(diaSemana) {
    let diaSemanaTextoSC

    switch (diaSemanaSC) {
        case 0:
            diaSemanaTextoSC = 'Domingo';
            return diaSemanaTextoSC;
        case 1:
            diaSemanaTextoSC = 'Segunda';
            return diaSemanaTextoSC;
        case 2:
            diaSemanaTextoSC = 'Terça';
            return diaSemanaTextoSC;
        case 3:
            diaSemanaTextoSC = 'Quarta';
            return diaSemanaTextoSC;
        case 4:
            diaSemanaTextoSC = 'Quinta';
            return diaSemanaTextoSC;
        case 5:
            diaSemanaTextoSC = 'Sexta';
            return diaSemanaTextoSC;
        case 6:
            diaSemanaTextoSC = 'Sábado';
            return diaSemanaTextoSC;
        default:
            return console.log('Erro');
    }
}

console.log(getDiaSemanaTexto(diaSemanaSC));

// ou

const diaFunction = getDiaSemanaTexto(diaSemanaSC);
console.log(diaFunction);

```

```

/* Aula 45 - Mais diferenças entre var, let e const

const verdadeira = true;

let nome = 'Jean';
var nome2 = 'Carlos';
console.log(nome, 'e', nome2);

//_ No mesmo escopo não se redeclara let, mas var pode.
//_ let tem escopo de bloco {... bloco}
//_ var só tem escopo de função

if(verdadeira){
    let nome = 'outro nome';
    var nome2 = 'nome redeclarado';
//_ pode declarar com o mesmo nome, pois no bloco let tem outro escopo, po
rtanto dentro é outra variável
    console.log(nome, 'e', nome2);
}
//_ se o bloco se depara com uma variável let, tentará buscar no bloco, e
irá voltando até o escopo global.
//_ já o var, mesmo se for usado em blocos, estará redeclarando

console.log(nome, 'e', nome2);

/*
    _ funções tem blocos especiais que são isolados. Suas informações não
podem ser vistas de fora.
    _ mesmo var se torna isolada, mas a função pode acessar variáveis e da
dos de fora.
*/

//! hoisting -> elevação

console.log(hoistingVar);    //? resultado -> undefined

var hoistingVar = 1;
//_ aqui ele eleva a declaração, mas sem o valor definido dela.

console.log(hoistingLet);    //? erro
//let hoistingLet = 1;      //? resultado -> is not defined
//_ não ocorre a elevação de declaração nesse caso
/* Aula 46 - Atribuição via desestruturação (Arrays)

```



```

//? ex:
let a = 'A';    //B
let b = 'B';    //C
let c = 'C';    //A

const letras = [b, c, a];
[a, b, c]=letras;

console.log(a, b, c);

//| mais detalhes

const num = [1, 2, 3, 4, 5, 6, 7, 8, 9];
//const primeiroNum = num[0];

const [primeiro, , terceiro, ...resto] = num;
console.log(primeiro, terceiro, resto);
/*
    #nota#
    _ A desestruturação pega como se fosse as posições, comendando no pri
meiro índice.
    _ e atribui a cada elemento contido no array a esquerda.
    _ se quiser pular valores precisa deixar espaços vazios.
    _ O operador ...(nomeDaVarivel) faz com que tudo que não foi atribuind
o ainda
    _ seja salvo na variável.
    _ Todos os elementos citados no array a esquerda terão o tipo que foi
usado para declarar,
    _ podendo ser let ou const (no caso todos são const)
    _ o operador ... tem o nome de rest (rest operator),
    _ se usar em contexto diferente pode se chamar spread
*/

//| atribuição por desestruturação com arrays multiplos

const num2 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];
console.log(num2[1][2]);    //? resultado -> 6
//_ Para acessar elementos de um array dentro de um array.

const [, [, , seis]] = num2;
console.log(seis);

//* Aula 47 - Atribuição via desestruturação (objetos)

```

```

const pessoa = {
  nome: 'Jean Carlos',
  sobrenome: 'De Meira',
  idade: 23,
  endereco: {
    rua: 'Rua legal',
    numero: 4,
  },
}

//_ Atribuição via desestruturação
//_ É possível setar valor padrão. Se não tiver valor é usado o padrão.
//_ Pode-
se usar uma atribuição com nome diferente usando o exemplo de nome abaixo.
//_ também é possível usar o rest ...
const {nome: teste = 'Não encontrado', sobrenome, idade, ...rest } = pessoa;
console.log(teste, sobrenome, idade, rest);

const {endereco: {rua, numero:num}, endereco}= pessoa;
console.log(rua, num, 'e', endereco);

/* Aula 48 - for clássico - estrutura de repetição

//#Nota# tais estruturas fazem ações repetidas, para evitar repetição de código.

console.log('Linha 0');
console.log('Linha 1');
console.log('Linha 2');
console.log('Linha 3');
console.log('Linha 4');
console.log('Linha 5');

for(let i=0; i<=5; i++){
  console.log(`Linha ${i} com repetição for`);
}

/*
  _ Estruturas de repetição precisam de uma variável de controle, normalmente usado o i
  _ Pois irá representar a palavra index, que significa índice. Ela precisa ter valor inicial
  _ Então é preciso usar uma comparação, pois a estrutura repete o código enquanto a pergunta
  _ retornar true (verdadeiro).

```

```

    _ E por fim precisa de um incremento ou decremento, pois precisa ter m
    udanças no valor da variável
    _ de controle, se não a repetição é infinita.
    _ o i ou variável de controle pode começar de qualquer valor e a alter
    ação pode ser de um em um ou de qualquer valor
    _ em qualquer valor.
    _ é possível iniciar i com valores negativos.
*/

```

```
const frutas = ['maçã', 'pêra', 'uva', 'Manga', 'laranja', 'Mamão'];
```

```
for(let i=0; i<frutas.length; i++){
    console.log(`Índice ${i}: `,frutas[i]);
}
```

```
/* Aula 50 - DOM e a árvore do DOM
```

```

/*
    _ window é o elemento pai de todos. -> é a tela
    _ depois temos o document    -> é o documento html
    _ então tem o HTML em si
    _ que contém o head e body por exemplo
    _ depois tem diversos elementos filhos
    _ Tendo os mais diversos elementos possíveis.
*/

```

```
/* Aula 51 - For in - estrutura de repetição
```

```
const frutas = ['Pêra', 'Maça', 'uva'];
const pessoa = {
    nome: 'Jean',
    sobrenome: 'Meira',
    idade: 23,
}
```

```
//#nota# o for in lê os índices ou chaves.
```

```
for(let indices in frutas){
    console.log(frutas[indices])
}
```

```
for(let chave in pessoa){
    console.log(chave,':', pessoa[chave]);
}
```

## /\* Aula 52 - for of - estrutura de repetição

```
const nome = ['Jean', 'Carlos', 'De Meira'];
```

```
for(let i=0; i<nome.length; i++){
  console.log('for clássico: ', nome[i]);
}
```

```
console.log('#####');
```

```
for(let i in nome){
  console.log('for in:', nome[i]);
}
```

```
console.log('#####');
```

```
//_ Já acessa diretamente o valor
```

```
for(let valor of nome){
  console.log('for of:', valor)
}
```

```
console.log('#####');
```

```
//obs# os acima podem funcionar com uma string, iterando sobre as letras
ou elementos da string
```

```
//obs# o forEach não consegue iterar sobre uma string
```

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

```
//_ para objetos pode diferenciar um pouco o for in funciona e quanto aos
demais será falado
```

```
//_ mais detalhes adiante. Pode avaliar como não iterável, pois não associa
a com índices.
```

```
//_ For clássico - geralmente para iteráveis (arrays e strings)
```

```
//_ for in - retorna o índice ou chave (string, arrays e objetos)
```

```
//_ for of - retorna o valor em si (iteráveis, como arrays e strings)
```

## /\* Aula 54 - while e do while - estruturas de repetição

```
/// while
```

```
/*
```

```
_ no while a variável de controle é criada fora do laço, a condição é
colocada na criação do while e
```

\_ e a atualização da variável de controle é feita dentro do corpo do código do while.

! Não esquecer de atualizar, pois se não cria condições de laços infinitos. Onde a repetição nunca acaba;

```
*/

let i = 0;

while(i <= 10){
  console.log(i);
  i++;
}

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

const min = 1;
const max = 30;
let rand = random(min, max);
console.log(rand);
//? rand = 10; //comente e descomente essa linha para ver o efeito de diferença do while e do do while

while(rand !== 10){
  rand = random(min, max);
  console.log('While - Número aleatório é:',rand);
}

//#nota# usado geralmente quando não se sabe quantas vezes o laço deve ser executado.

//| do while
/* //#nota#
  _ a diferença para o while é que o while checa a condição e depois executa o código se a condição for verdadeira.
  _ podendo assim não executar nenhuma vez se a condição for falsa logo de cara.
  _ já o do while executa primeiro e depois testa a condição, se a condição for verdadeira vai executar novamente e
  _ testar ao final de cada execução, assim sendo mesmo que a condição seja verdadeira de cara, o código irá executar pelo
  _ menos uma vez.
*/
```

```

console.log('#####');
do{
    rand = random(min, max);
    console.log('do while - Número aleatório é:',rand);
}while(rand!==10);

/*  //#nota#
    _ Comente e descomente a linha indicada acima, então o comportamento d
e while e do while serão mais evidentes.
    _ Ao descomentar a linha, o valor inicial de rand será 10, que fará co
m que a condição seja inicialmente verdadeira
    _ então do while não irá executar;
    _ Mas o do while sim, e dentro do mesmo o rand será chamado, fazendo o
s valores seres aleatórios novamente; e assim executando
    _ o código até encontrar um valor igual a 10.
*/

//* Aula 55 - Break e continue

//#nota# funciona em todos os laços, ajuda a permitir o controle de quando
pular um elemento ou
//_ quebrar o laço e sair a qualquer momento que quiser.

const numeros =[1, 2, 3, 4, 5, 6, 7, 8, 9];

for(let numero of numeros){

    if(numero === 2 || numero === 5){ //? pode ter mais condições
        console.log('Pulei o número');
        continue;
        //#nota# sempre que achar continue ele pula para próxima interação
do laço
    }
    if(numero === 8){
        console.log('Pulei o número');
        continue; //? pode ter mais de um continue
    }
    console.log(numero);
    if(numero === 7){
        console.log('encontrei o número 7, saindo. ');
        break;
        //#nota# o break força a saída do laço, ou da estrutura sendo usada.
    }
}
}

```

```
//#nota# Cuidado com while e do while, pois a atualização da variável pode
acabar sendo interrompida
//#nota# por que o continue ia para próxima iteração, então o valor não mu-
daria mais, caindo em laço
//#nota# infinito. Portanto, se deve atuazar a variável também antes de um
continue.
```

### //Exercícios

```
// Escreva uma função que recebe dois números e retorna o maior deles
```

```
function maiorEntreDoisNumeros(num1, num2){
  return num1 > num2 ? num1 : num2
}
```

```
console.log(maiorEntreDoisNumeros(2,10));
```

```
//? usando arrow function de retorno de uma linhas
```

```
const max2 = (num1, num2)=> num1 > num2 ? num1 : num2;
console.log(max2(21,1));
```

```
/*
```

```
_ Escreva uma função chamada ePaisagem que recebe dois argumentos, lar-
gura e altura
_ de uma imagem (number).
_ Retorne true se a imagem estiver no modo paisagem.
```

```
*/
```

```
const ePaisagem = (largura, altura)=> largura > altura ? true : false;
console.log(ePaisagem(21,1));
```

```
//? por retornar apenas true e false é possível omitir o ternário,
//? então ele checa a condição e retorna o resultado.
```

```
const ePaisagem2 = (largura, altura)=> largura > altura;
console.log(ePaisagem2(21,1));
```

```
/*
```

```
Escreva uma função que recebe um número
retorne o seguinte:
```

```
Número é divisível por 3 = Fizz
```

```
Número é divisível por 5 = Buzz
```

```
Número é divisível por 3 e 5 = FizzBuzz
```

```
Número não é divisível por 3 e por 5 = Retorna o próprio número
```

```
Checar se o número é realmente um número = retorna o próprio número
```

```
Use a função com número de 0 a 100
```

```

*/
const result = num => {
  if(num === 0) return num;

  if(num > 100 || num < 0){
    return 'fora dos limites';
  }else{ if(typeof(num) === "number"){
    if(num % 3 === 0 && num % 5 === 0) return 'FizzBuzz';
    if(num % 3 === 0) return 'Fiz';
    if(num % 5 === 0) return 'Buzz';
    return num;
  }else{
    return 'Não é um número'
  }
}
}

for(let i=0; i<=100; i++){
  console.log(result(i));
}

/* Aula 59 - Tratando e lançando erros (try, catch e throw)

//_ try é como um tente, e se der erro cairá no catch

// try{
//   console.log(naoExisto); //? contém códigos que podem dar erros.
// }catch(err){
//   console.log('naoExisto não existe');
//   console.log(err);
// }

// #Obs# a execução para no 1º erro, então comentar e descomentar o código
para analisar as ocorrências.

// #Nota# não é recomendado exibir o erro para o usuário, pois isso é poten
cialmente perigoso para sua aplicação

//| Lançando erros;

function soma(x, y){
  if(typeof x !== 'number' || typeof y !== 'number'){
    throw('X e y precisam ser números')
  }
  return x + y;
}

```



```

function soma2(x, y){
  if(typeof x !== 'number' || typeof y !== 'number'){
    throw new Error('X e y precisam ser números')
  }
  return x + y;
}

try{
  console.log(soma2(1,2));
  //console.log('com tryCatch',soma('a',2));  //? descomentar para usar
  função soma, então o erro só retornará a string.
  console.log(soma2('a',2));
}catch(error){
  console.log(error);
}

//#nota# Ao usar a soma2 também é criado um erro, então esse log de erro v
olta ao apresentar o erro, diferente da função soma.
//#nota# se criar um ReferenceError, será retornado no console um Referenc
eError. Existem vários tipos de classes de error.
//#Importante# Não retorne o erro para o usuário, aqui está sendo retornad
o por ser ambiente de estudo.

/* Aula 60 - Tratando e lançando erros (try, catch e finally)

try{
  console.log('Abri um arquivo');
  console.log('Manipulei o arquivo');
  console.log('gerou erro');
  console.log(a);
  console.log('Fechei o arquivo');
  //? Executada quando não há erros.
}catch(e){
  console.log('Tratando o erro');
  //? executada quando há erros.
}finally{
  console.log('Fechar o arquivo, Finally');
  //? sempre é executado.
}

//#Nota# podem ter tryCatches aninhados (um dentro do corpo de código do ou
tro.)

//| parte 2

```

```

function retornaHora(data){
  if( data && !(data instanceof Date)){
    throw new TypeError('Esperando instância de Date.');
```

```

  }

  if(!data){
    data = new Date();
  }
  return data.toLocaleTimeString('pr-Br',{
    hour: '2-digit',
    minute:'2-digit',
    second:'2-digit',
    hour12: false,
  });
}
```

```

try{
  const data = new Date('01-01-1970 12:58:12');
  const hora = retornaHora();
  console.log(hora);
}catch(erro){
  // Tratar error
  console.log(erro)
}finally{
  console.log('Tenha um bom dia');
}
```

```

/*
  #Nota#
  _Nesse programa se receber a const data, irá executar com a data unix
e o finally com o
  _tenha uma bom dia. Se for enviado vazio retornará a hora atual e o fi
nally.
  _ Se for enviado formatos de dados diferentes será gerado um erro com
o throw e cairá no catch,
  _que nesse caso mostra o erro, e então cairá no finally também.
```

```

*/
```

```

//* Aula 61 - setInterval e setTimeout
```

```

function mostraHora(){
  let data = new Date();
  return data.toLocaleTimeString('pt-Br',{
    hour12: false,
```

```

    })
}

const timer = setInterval(function(){
    console.log(mostraHora());
},1000);
/*
    #Nota#
    _ A função setInterval vai configurar o intervalo de tempo para que al
guma função
    _ seja executada em determinado tempo.
    _ setInterval recebe dois parâmetros, o primeiro é qual função será exe
cutada
    _ O segundo é de quanto em quanto tempo será executada em miliSegundos
(mS).
    _ A função executada para setInterval não pode retornar valor.
*/

setTimeout(function(){
    clearInterval(timer)
}, 5000);

/*
    #Nota#
    _ A função setTimeout executará uma vez só, assim que o tempo setado p
assar.
    _ Também recebe dois parâmetros, cujo o primeiro é a função que será e
xecutada, e o segundo
    _ É o tempo para ser executado.
    _ Também precisa de uma função sem retorno.
    _ clearInterval vai interromper a execução.
*/

//Funções avançadas

/* Aula 64 - Maneiras de declarar funções

//| declaração
function falaOi(){
    console.log('Oi');
};
//_ É uma declaração mais literal.
//_ Ocorre o function hoisting -> eleva a declaração.
//_ declaração mais clássica.

```

```

falaOi();

/// functions expressions
//_ função (todas) são first-class objects (objetos de primeira classe)
//_ podem ser tratadas como dado.

const souUmDado = function(){
    console.log('Sou um dado');
}

souUmDado();
//_ Pode executar a constante ou variável como uma função.
//_ Permite passar funções como parâmetros.

function executaFuncao( funcao){
    funcao();
}

executaFuncao(souUmDado);

/// Arrow function -> function expression resumida na declaração

const arrow = ()=> console.log('Sou uma arrow function');

arrow();

/// Dentro de objetos

const obj = {
    //falar: function(){    //? abaixo tem o modo mais novo de declarar um
    método do objeto.
    falar(){
        console.log('Estou falando...');
    }
}

obj.falar();

/* Aula 65 - Parâmetros da função

function funcao(){
    console.log(arguments);
}

/*

```

```

#Nota#
_ Funções criadas com a palavra function tem um parâmetro com nome arguments, que guarda(sustenta) todos os argumentos.
_ Mesmo que não crie parâmetros a função vai salvar os argumentos inviados para suprir os parâmentros.
_ arguments é um objeto.
*/

funcao(1, 2, 3, 4, 5, 6, 7);

function funcao2(a=0, b=0, c=0, d=0,e=0){
    console.log(arguments);
    console.log(a, b, c);
}

funcao2(1, 2, 3,undefined, 5, 6, 7);
/*
#Nota#
_ Isso ainda é válido caso existam argumentos criados. arguments ainda sustentará todos os argumentos.
_ Se tiver parâmetros, mas não for passado argumentos para todos, o parâmetro será criado
_ mas tera valor de undefined.
_ É possível usar valores pré definidos para os parâmetros, e se os mesmos não receberem argumentos
_ adotaram o valor inicial indicado.
_ Se mandar undefined ele também adotará o valor padrão, Apesar de isso não ser muito indicado.

_ arguments não funciona com arrow functions.
_ É possível usar a desestruturção de objetos ou arrays dentro dos parâmetros de uma função
_ Rest operator se for usado tem que ser sempre o último parâmetro da função;
*/

/** Aula 66 - Retorno de uma função
*/

#Nota#
_ return retorna algum valor
_ encerra a função
_ existe um retorno undefined por padrão
_ se não existir a palavra return terá um retorno undefined
_ funções podem ou não retornar valores. (além do undefined)

```

```

*/

function soma(a, b){
    return a+b;
}

console.log(soma(5,2));

//| retornando uma função sem executar imediatamente

function criaMultiplicador(multiplicador){
    //multiplicador está aqui

    /*
        _ Abaixo poderia ser de duas formas
        _ poderia ser criado a função e depois
        _ retornar a mesma sem () e argumentos,
        _ para que assim se retorne a função,
        _ que será executada em outra hora.
        _ ou retornar diretamente a função
        _ que assim também será executada depois.
    */
    /*
    // function multiplica(n){
    //     return n * multiplicador;
    // }
    // return multiplica;

    return function(n){
        return n * multiplicador;
    };
}

const duplica = criaMultiplicador(2);
const triplica = criaMultiplicador(3);
const quadriplica = criaMultiplicador(4);

console.log(duplica(2));
console.log(triplica(2));
console.log(quadriplica(2));

/* Aula 67 - Escopo léxico

const nome = 'Jean';

function falaNome(){

```

```

    console.log(nome);
}
falaNome();
/*
    #nota#
    _ A função reconhece o que está declarado em torno dela,
    _ voltando escopos até chegar no global
*/

```

### /\*\* Aula 68 - Closures

```

/*
    #Nota#
    _ relacionado ao escopo léxico
    _ a const funcao tem acesso a três escopos.
    _ o dela, o da função mãe (retornaFuncao) e o global.
    ! O closure é a habilidade da função de acessar ao seu escopo léxico
    _
*/

function retornaFuncao(){
    const nome = 'Luiz';
    return function(){
        return nome;
    }
}

const funcao = retornaFuncao();
console.log(funcao);    //? tem uma função anônima
console.dir(funcao);    //? no browser terá diferenças, inclusive mostra
os escopos acessíveis.

```

### /\*\* Aula 69 - Funções de callback

```

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

function f1(callback) {
    setTimeout(function () {
        console.log('f1');
        if (callback) callback();
    }, 1000);
}

```

```

    }, rand());
}

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

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

f1(f1Callback);

function f1Callback() {
    f2(f2Callback);
}

function f2Callback() {
    f3(f3Callback);
}

function f3Callback() {
    console.log('Olá mundo!');
}

/*
#Nota#
_ A função rand() e o setTimeout estão simulando interações com a web,
onde algo demoraria
_ um tempo indeterminado para realizar a ação.
_ Se não existisse o callback as funções poderia ser executadas em ord
ens aleatórias, e
_ isso pode comprometer o funcionamento de sistemas. Portanto é envid
o um callback como argumento
_ para as funções.
_ e colocado uma condições, para que, se existir um callback nos parâ
etros recebidos o mesmo
_ será executado, independente do que seja.

```



```

_ Para melhor organização cria-
se funções f_Callback(), para que não seja colocadas aninhadas uma
_ dentro da outra. Dessa forma se garante a ordem de execução das cois
as com o callback mesmo que
_ cada função demore tempos diferentes para serem executadas.
*/

```

## /\*\* Aula 70 - Funções imediatas (IIFE)

```

//? Ou funções auto invocadas
//_ IIFE -> Immediately Invoked Function Expression

/*
    #Nota#
    _ Quando criamos algo que roda na web ou em contextos as variáveis e f
unções que
    _ criamos pque podem tocar o escopo global podem acabar sendo usadas,
alteradas
    _ ou mesmo conflitar com já existentes no escopo global, é preferível
evitar que isso
    _ ocorra. Desse modo é possível proteger o nosso escopo o envolvendo e
m uma função.
    _ porém essa função vai todar o escopo global, e isso pode não ser a m
elhor opção.
    _ As funções IIFE são anônimas e auto invocadas, dessa forma não é pos
sível de
    _ serem chamada, não tocando o escopo global assim.
    _ A estrutura da IIFE é envolvida em () e após terminar é colocado ()
para chamar a mesma.
    !   Exemplo da estrutura
    !   (function(){
    !       CORPO DA FUNÇÃO
    !   })();
*/

(function (idade) {
    const nome = 'Jean Meira'
    console.log(nome, ':', idade);
})(23);

const nome = 'Qualquer coisa';

```

## /\*\* Aula 71 - Funções fábrica (factory functions)

```

function criaPessoa(nome, sobrenome, altura, peso){
  return{
    nome,
    sobrenome,
    //getter
    get nomeCompleto(){
      return `${this.nome} ${this.sobrenome}`
    },
    //setter
    set nomeCompleto(valor){
      valor = valor.split(' ');
      this.nome = valor.shift();
      this.sobrenome = valor.join(' ');
    },
    fala(assunto){
      return `${this.nome} está falando sobre ${assunto}`
    },
    altura,
    peso,
    get imc(){
      const indice = this.peso/(this.altura**2);
      return indice.toFixed(2);
    },
  };
}

const p1 = criaPessoa('Jean', 'Meira', '1.72', '60');
console.log(p1.fala('Js'));
console.log(p1.imc);
p1.nomeCompleto = 'Jean Carlos De Meira' //? com o setter podesse alterar
diretamente na chave.
console.log(p1.nomeCompleto);
/*
  #Nota#
  _ o this referência quem chamou a função,
  _ no contexto acima o this.nome é quase o mesmo que p1.nome
  _ colocar o get imc(), faz parecer que ele é um atributo normal.
  _ o get vem de getter e set vem de setter.
  _ Factory function criam e retorna os objetos
  _ isso permite diminuir o código,
*/

/* Aula 73 - Funções construtoras (constructor functions)
*/

```

```

#Nota#
_ Funções construtoras retornam objetos (constroem)
_ são iniciadas com letras maiúsculas. e obrigatoriamente contém a palavra new.
_ new cria um novo objeto vazio, faz o this apontar para o objeto vazio, e retorna
_ implicitamente.
_ É possível ter atributos, métodos e variáveis privadas, que estão acessíveis apenas
_ dentro da função construtora.
_ Ao usar o this. cria-se atributos e métodos públicos que podem ser acessados pela notação
_ de . fora da função construtora.
_ A exemplo de p1.nome, p2.nome e p1.metodo demonstrados abaixo.
*/

```

```

function Pessoa(nome, sobrenome){
  //_ Atributos ou métodos privados
  const ID = 123456;
  const metodoInterno = function(){

  };

  //_ Atributos ou métodos públicos
  this.nome= nome;
  this.sobrenome = sobrenome;
  this.metodo= function(){
    console.log(this.nome + ': Sou um método');
  }
}

```

```

const p1 = new Pessoa('Jean', 'Meira');
const p2 = new Pessoa('João', 'André')
console.log(p1, p2);
console.log(p1.nome, p2.nome);
p1.metodo();

```

/\* Aula 75 - Funções recursivas

```

/*
#Nota#
_ É uma função que ela mesma se chama.
_ cuidado com o máximo de recursividade, em certos valores altos vai dar erro
_ quase como um loop infinito.

```

```

*/

function recursiva(max){
  if(max < 0) return;
  if(max > 10) return;
  console.log(max);
  max++;
  recursiva(max);
}

recursiva(0);

```

## ``` /** Aula 76 - Funções geradoras ```

```

/*
  #Nota#
  _ Funções geradoras não vão retornar todos os valores de uma vez.
  _ é quase como se houvesse um pause ao longo de seu código
  _ usa um prencípio de lazy evaluation (avaliação preguiçosa)
  _ e pode ser bom por causa de performance em alguns casos.
  _ Tem um método incluso, o next(), que retorna o valor (value)
  _ e se já acabou o gerador (done);
  _ A função é iterável;
  _ A função geradora normalmente usa p yield para retornar valores
  _ que funciona como um retorno fracionado, a cada vez que chama retorn
a a
  _ parte que está na lista dentro do corpo da função, repetindo isso at
é o último
  _ yield disponível. Se usar o return irá retornar o valor, mas também
irá quebrar
  _ a sequência disponível, fazendo com que a próxima chama apresente er
ros.
*/

function* geradora1() {
  //código qualquer ...
  yield 'Valor 1';
  //código qualquer ...
  yield 'Valor 2';
  //código qualquer ...
  yield 'Valor 3';
}

const g1 = geradora1();
console.log(g1.next().value);

```

```
console.log(g1.next().value);
console.log(g1.next().value);

const g2 = geradora1();
for(let valor of g2){
    console.log('For: ',valor);
}

function* geradora2(){
    let i = 0;
    while(true){
        yield i;
        i++;
    }
}

const g3 = geradora2();
console.log(g3.next().value);
console.log(g3.next().value);
console.log(g3.next().value);

function* geradora3(){
    yield 0;
    yield 1;
    yield 2;
}

function* geradora4(){
    yield* geradora3();
    yield 3;
    yield 4;
    yield 5;
}

const g4 = geradora4();
for(let valor of g4){
    console.log('delegando função', valor);
}

function* geradora5(){
    yield function(){
        console.log('Vim de y1');
    };

    yield function(){
        console.log('Vim de y2');
    };
}
```

```

    }
}

const g5 = geradora5();
const func1 = g5.next().value;
const func2 = g5.next().value;
func1();
func2();

```

//Arrays Avançado

/\* Aula 077 - Revisão do básico de Arrays

```

/*
    #Nota#
    _ são tipos de dados por referências, se quiser copiar tem que usar o
    _ spread operator [...(nomeDoArrayASerCopiado)]. É o mesmo operador de
    _ rest operator, mas usado em contextos diferentes.
    _ delete (array)[posição], deleta o elemento sem mover as posições.
    _ .pop() remove o último item
    _ .shift() remove o primeiro. Ambos podem atribuir o item removidos a
    variáveis.
    _ .push() adiciona elemento no final do array.
    _ .unshift() adiciona elementos no início do array.
    ! Não é comum a adição e remoção no início do array, pois desloca toda
    s as posições.
    ! Isso pode ser não performático para arrays maiores.
    _ .slice() pega elementos indicados em uma posição início até uma posi
    ção final, sem
    _ contar a última posição. Pode ter números negativos na segunda posiç
    ão (a de término)
    _ então conta quais itens serão removidos nesse caso. -1 remove 1, -
    2 remove 2, etc;
    _ .split() separa uma string por algo em comum, se inserir o espaço, s
    epara a cada espaço,
    _ o resultado é um array.
    _ .join() uni um array, é indicado dentro do método qual será o separa
    dor, que estará
    _ entre um elemento e outro na formação do resultado final, que é uma
    string.
*/

const nomes = ['Eduardo', 'Maria', 'Joana','wallace', 'rosana'];
console.log(nomes);

```

```

//? é possível usar para string, objetos, funções, números, criar usando o
construtor;

const nomes2 = new Array('Eduardo', 'Maria', 'Joana');
delete nomes2[2]
console.log(nomes2);

const novo = nomes.slice(1,3);
//_ possível usar negativos, daí trata-se de quantos serão removidos, -
1 remove 1, -2 remove 2 etc
console.log(novo);

//| separar string em arrays.

const nome = 'Jean Carlos De Meira';
const meuNome = nome.split(' ');
console.log(meuNome);
const nomeUnido = meuNome.join(' ');
console.log(nomeUnido);

/* Aula 078 - Método splice

/*
    #Nota#
    _ .splice(indíce qualquer, elementos deletados, elementos para adicio
nar('name1', 'name2'...))
    _ aceita negativos, que começa de modo invertido (referencia a posição
do índice menos
    _ o valor total de elementos do array)
    _ se usar Number.MAX_VALUE no segundo número, o array começará em uma
posição e irá até
    _ a última posição.
    _ Pode usar 0 no segundo número, fazendo com que não delete nada.

*/

const nomes = ['maria', 'joão', 'eduardo', 'gabriel', 'julia'];

const removidos = nomes.splice(0, 1, 'Jean');

console.log(nomes, removidos);

const nomes2 = [...nomes]
nomes2.splice(2, Number.MAX_VALUE);
console.log(nomes2);

```

```

nomes.splice(nomes.length,0,'Luiz');
console.log(nomes);

//| Simulando métodos.
//_ Pop -> remove o último
nomes.splice(-1,1);
console.log(nomes);
//_ shift -> remove o primeiro
nomes.splice(0,1);
console.log(nomes);
//_ push -> adiciona um no final
nomes.splice(nomes.length,0, 'Jean');
console.log(nomes);
//_ unshift -> adiciona um no começo
nomes.splice(0,0,'luiz');
console.log(nomes);

/** Aula 079 - Concatenando arrays

/*
    #Nota#
    _ .concat() permite concatenar arrays, podendo ser eles alocados em v
    ariáveis/constantes
    _ arrays literais, ou mesmo array com algum elemento.
*/

const array1 = [1, 2, 3];
const array2 = [4, 5, 6];
const array3 = array1.concat(array2, [7, 8, 9], 'Jean');
console.log(array3);

const array4 = [...array1, ...array2, 'Jean', ...[7, 8, 9]];
console.log(array4);

/** Aula 080 - Filter - filtrando o array

/*
    #Nota#
    _ filter é somente para arrays.
    _ Não edita o valor do array original.
    _ Ela recebe a função de callback, que irá iterar sobre todas as posiç
    ões do array.
    _ a função de callback deve retornar true ou false, true quando o elem
    ento deve ser adicionado

```



```

    _ ao novo array, e false quando não deve.
    _ Se a função for usada apenas por um filter e não se repetir em mais
lugares, é mais interessante
    _ usar dentro do filter() como uma função anônima. Ou ainda uma arrow
function.
    _ os parametros que não são usados podem ser omitidos também.
    _ O return é de apenas uma linha, também pode ser simplificado, omitin
do a palavra return
    _ e tirando as chaves do código.
    _ exemplo:
    _      .filter((valor)=> valor > 10)
    ! filter retorna sempre um array, com a mesma quantidade de elementos
do original ou menos.
*/

/// exemplo 1
// retorne os números maiores que 10
const numeros = [5, 50, 80, 1, 2, 3, 5, 8, 7, 11, 15, 22, 27];

function callbackFilter(valor, indice, array){
    return valor > 10;
}

const numerosFiltrados = numeros.filter(callbackFilter);
console.log(numerosFiltrados);

/// exemplo 2
// retorne as pessoas que tem o nome com 5 letras ou mais
// retorne as pessoas com mais de 50 anos.
// retorne as pessoas cujo nome termina com a.

const pessoas =[
    { nome: 'Luiz', idade: 62},
    { nome: 'maria', idade: 23},
    { nome: 'eduardo', idade: 55},
    { nome: 'Letícia', idade: 19},
    { nome: 'rosana', idade: 32},
    { nome: 'wallace', idade: 47},
    { nome: 'Jean', idade: 23}
];

const pessoasComNomeGrande = pessoas.filter((valor)=> valor.nome.length >=
5);
console.log(pessoasComNomeGrande);

```

```

const pessoasComMaisDe50Anos = pessoas.filter((valor)=> valor.idade > 50);
console.log(pessoasComMaisDe50Anos);

const pessoasComNomeTerminadosEmA = pessoas.filter((valor)=> {
  return valor.nome.toLowerCase().endsWith('a');
})
console.log(pessoasComNomeTerminadosEmA);

/* Aula 081 - map - mapeando o array

/*
  #Nota#
  _ Map() altera os valores do array original.
  _ retorna um array com exatamente o mesmo número de posições do original.
  _ só que com valores alterados.
  _ também funciona como filter, recebe índice, posição e array,
  _ precisa de uma função de callback.
  _ retorna em um novo array
  ! Cuidado que há casos que podem alterar o objeto original.
  ! Se isso for desejado é bom, caso não usar o spread para criar
  ! Objetos intermediários.
  _ A única mudança é o retorno, que quer mudar os valores.
*/

//| exemplo 1
// dobre os números
const numeros = [5, 50, 80, 1, 2, 3, 5, 8, 7, 11, 15, 22, 27];
const dobro = numeros.map((valor, indice, array)=>{
  return 2*valor;
});
console.log(dobro);

//#Obs# é possível retornar de modo simplificado se houve apenas uma linha
de retorno.
const triplo = numeros.map((valor, indice, array)=> 3*valor);
console.log(triplo);

//| exemplo 2
// retorne apenas uma string com o nome das pessoas.
// remova apenas a chave 'nome' do objeto
// adicione uma chave id em cada objeto.

const pessoas =[
  { nome: 'Luiz', idade: 62},

```

```

    { nome: 'maria', idade: 23},
    { nome: 'eduardo', idade: 55},
    { nome: 'Letícia', idade: 19},
    { nome: 'rosana', idade: 32},
    { nome: 'wallace', idade: 47},
    { nome: 'Jean', idade: 23}
  ];

const nome = pessoas.map((valor)=> valor.nome);
console.log(nome);

const idades = pessoas.map((valor)=>{
  //delete valor.nome;
  //return valor;
  //_ podia ser também o retorno de um novo objeto.
  return {idade: valor.idade};
})
console.log(idades);
  //! Ao usar o primeiro caso do exemplo acima ele deletou na referência
  //! com isso alterou o array original.

const comIds = pessoas.map((valor, index)=>{
  const novoObj = {...valor}
  novoObj.id = index;
  return novoObj;
})
console.log(comIds);

  //!Aqui o mesmo caso pode ocorrer, para evitar isso foi criado um novo
  //! objeto, para que o original não fosse alterado.

console.log(pessoas);

/* Aula 082 - reduce - reduzindo um array

/*
#Nota#
_ reduce() é mais utilizado para reduzir o objeto a um único objeto.
_ reduce é muito versátil, ele até consegue imitar o map e filter,
_ mas não é ideal
! não faça isso, use cada um pelo seu funcionamento mesmo.
_ reduce() recebe os argumentos iguais ao map e filter, mas também rec
ebe

```

```

    _ um acumulador. O acumulador pode receber um valor inicial, como indicado a seguir.
    _ exemplo
    _ xxxx.reduce(function(acumulador, valor, indice, array){ corpo do código},0 {<-valor inicial})
    _ a cada iteração a função deve retornar algo, se não o acumulador passa a ser undefined
    ! cuidado com o valor do acumulador, se não houve valor indicado será adotado o primeiro valor
    ! do array. Isso pode implicar em erros indesejados.
    _ se não mandar valor em acumulador, ele será o primeiro e na primeira iteração o valor
    _ será o segundo elemento.
*/

```

```

//| exemplo 1
// some todos os números (reduce)
// retorne um array com os pares (filter)
// retorne um array com o dobro dos valores (map)
const numeros = [5, 50, 80, 1, 2, 3, 5, 8, 7, 11, 15, 22, 27];
const total = numeros.reduce(function(acumulador, valor, indice, array){
    acumulador += valor;
    return acumulador;
}, 0)
console.log(total);

// acumulador -> accumulator -> acc
const pares = numeros.reduce((acc, valor) => {
    if(valor % 2 === 0) acc.push(valor)
    return acc;
}, [])
console.log(pares);

const dobro = numeros.reduce((acc, valor) => {
    const dobro = 2 * valor;
    acc.push(dobro);
    return acc;
}, [])
console.log(dobro);

//| exemplo 2
// retorne a pessoa mais velha
const pessoas = [
    { nome: 'Luiz', idade: 62 },
    { nome: 'maria', idade: 23 },

```

```

    { nome: 'eduardo', idade: 55},
    { nome: 'Letícia', idade: 19},
    { nome: 'rosana', idade: 64},
    { nome: 'wallace', idade: 47},
    { nome: 'Jean', idade: 23}
  ];

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

/* Aula 083 - Filter + Map + Reduce

/*
  #Nota#
  _ É possível encadear os métodos, ao invés de usar um por um criando novas variáveis.
  _ Ao executar um método o resultado será passado para o próximo que está encadeado.
*/

//| exemplo 1
// -> filtrar pares
// -> dobrar os valores
// -> reduzir (somar todos)
const numeros = [5, 50, 80, 1, 2, 3, 5, 8, 7, 11, 15, 22, 27];
const pares = numeros.filter((valor)=> valor % 2 === 0)
const dobro = pares.map((valor)=> valor*2)
const soma = dobro.reduce((acc, valor)=> acc+=valor)

console.log(pares);
console.log(dobro);
console.log(soma);

const resultado = numeros
  .filter((valor)=> valor % 2 === 0)
  .map((valor)=> valor*2)
  .reduce((acc, valor)=> acc+=valor)
console.log(resultado);

```

```

/** Aula 084 - ForEach

/*
  #Nota#
  _ Somente disponível em arrays;
  _ Somente itera sobre os valores.
  _ recebe uma função de callback
  _ Também recebe o valor, índice e array.
  _
*/

const array1 = [1, 2, 3, 4, 5, 6, 7, 8, 9];

array1.forEach((valor, index, array)=>{
  console.log(valor, index)
})

let total = 0;
array1.forEach( (valor)=>total += valor);
console.log(total);

//Objetos prototypes

```

```

/** Aula 85 - Revisando objetos / menção do freeze

/*
  #Nota#
  _ Na declaração de objeto de forma literal, contém as {} que envolvem
o corpo
  _ do objeto. Onde o objeto é literalmente escrito ali na declaração.
  _ Os objetos tem seu corpo formado por pares de chave e valor, separad
os por: entre si.
  _ para inserir novos conjuntos de par chave e valor se separa por vírg
ula ao final de cada par
  _ é possível acessar uma chave do objeto por notação de ponto -> pesso
a.nome ou com
  _ notação de colchete -> pessoa['nome']
  _ a notação de colchete aceita um uso dinâmico.
  _ é possível deletar uma chave com o delete
  _ ex -> delete pessoa1.nome
  _ objetos podem conter métodos.
  _ é possível usar factory functions e constructor functions;
*/

//| literal

```

```

const chave = 'nome';
const pessoa = {
  nome: 'Jean Carlos',
  sobrenome: 'De Meira'
}

///  
 Exemplo de uso dinâmico
console.log(pessoa[chave]);

///  
 construtor
const pessoa1 = new Object();
pessoa1.nome = 'Jean Carlos';
pessoa1.sobrenome = 'De Meira';
pessoa1.falarNome = function(){
  return (`${this.nome} está falando seu nome.`);
}
console.log(pessoa1.nome, pessoa1.sobrenome);
console.log(pessoa1.falarNome());

///  
 Iterar no objeto.
for(let chave in pessoa1){
  console.log(chave, pessoa1[chave]);
}

///  
 Factory function

function pessoaFac(nome, sobrenome){
  return{
    nome,
    sobrenome
  }
}

const p1 = pessoaFac('Jean Carlos', 'De Meira');
console.log(p1);

///  
 Constructor function

function PessoaCons(nome, sobrenome){
  this.nome = nome;
  this.sobrenome = sobrenome;

  Object.freeze(this); ///  
 Trava todos os objetos criados por esse molde
  .

  ///  
 Não permite mais nada além do já criado, não é possível alterar
  ///  
 criar ou deletar coisas no objeto.

```

```

    //_ literalmente congela o objeto.
    //! também funciona com arrays.
}

//_new cria uma objeto vazio {} <- atrelar o this ao objeto. e retorna já
o this que representa o objeto.
const p2 = new PessoaCons('Jean Carlos','De Meira');
console.log(p2);

/// travar mudanças nos valores da referência do objeto.

Object.freeze(p1);
//_ Não permite nem mudanças nos valores das chaves.

/* Aula 86 - Object.defineProperty() e Object.defineProperties()

/*
    #Nota#
    _ enumerable      -> mostra a chave
    _ value            -> atribui o valor
    _ writable         -> pode alterar ou não o valor
    _ configurable     -> pode apagar, editar ou reconfigurar a chave
*/

function Produto(nome, preco, estoque) {
    //this.nome = nome;
    //this.preco = preco;
    //this.estoque = estoque;

    Object.defineProperties(this, {
        nome: {
            enumerable: true,
            value: nome,
            writable: true,
            configurable: true,
        },
        preco: {
            enumerable: true,
            value: preco,
            writable: true,
            configurable: true,
        }
    });

    Object.defineProperty(this, 'estoque', {

```



```

        enumerable: true,
        value: estoque,
        writable: false,
        configurable: false,
    });
}

const p1 = new Produto('Camiseta', 20, 3);
console.log(p1);
console.log(Object.keys(p1)); //? -> mostra as keys que são enumerable

/* Aula 87 - Getters e Setters

/*
    #Nota#
    _ É uma maneira de proteger a propriedade de certa forma
    _ get obtém o valor
    _ set atribui o valor.
    _ não usar o this para set, pode ocasionar loop infinito, o set tem o
this, que chama o objeto
    _ que acessa o set e encontra o this, que novamente chama o objeto...
    _ get e set fazem com que a propriedade tenha a interação externa redu
zida, ou pelo menos
    _ tenha um intermediário, que faz com que o valor do dado possa ter re
strições ou regras.
*/

function Produto(nome, preco, estoque) {
    this.nome = nome;
    this.preco = preco;
    let estoquePrivado = estoque
    Object.defineProperty(this, 'estoque', {
        enumerable: true,
        //value: estoque,
        //writable: false,
        configurable: false,
        get: function(){
            return estoquePrivado;
        },
        set: function(valor){
            if(typeof valor !== 'number'){
                throw new TypeError('Um valor numérico é esperado');
            }
            estoquePrivado = valor;
        }
    })
}

```

```

    });
}

const p1 = new Produto('Camiseta', 20, 3);
console.log(p1);
console.log(p1.estoque);
//p1.estoque = 'Qualquer coisa'

p1.estoque = 2;
console.log(p1.estoque);

/** Aula 88 - Métodos úteis para objetos
/*
#Nota#
_ Object.values
_ Object.entries
_ Object.assign(des, any)
_ Object.getOwnPropertyDescriptor(o, 'prop')
_ ...spread

#obs# já vimos
_ Object.keys      -> retorna as chaves
_ Object.freeze    -> congela o objeto
_ Object.defineProperties -> define várias propriedades
_ Object.defineProperty  -> define uma propriedade;

! Descrições
_ spread espalha o objeto, o que deslinca do mesmo lugar da memória
_ se puxar vários objetos com chaves iguais, vai manter o valor da última
_chave lida ao puxar os objetos.

_ Object.assign({}, produto) -> cria um objeto vazio e atribui o valor
de produto nele
_ pode aderir novos valores também, e é possível inserir mais de um objeto
já existente no mesmo

_ Object.getOwnPropertyDescriptor(o, 'prop') retorna os valores das propriedades,
que são
_ value: 'Caneca',
_ writable: true,
_ enumerable: true,
_ configurable: true

```

```

    _ Object.values
    _ retorna os valores das chaves

    _ Object.entries
    _ vai servir para iterar no arrays, mas também retorna cada par chave e
    valor em um array
    _ separado.

*/

//| copiando objetos.

const produto = {nome: 'Caneca', preco:1.8}
const outraCoisa = {...produto, material: 'Porcelana'}

outraCoisa.nome = 'Camisa';
outraCoisa.preco = 2.5;

console.log(produto);
console.log(outraCoisa);

const caneca = Object.assign({}, produto, {material: 'vidro'})
console.log(caneca);

//_ também é possível copiar de forma manual, cria uma chave e atribui dir
etamente o valor
//_ ex -> const exemplo = {nome: produto.nome}

console.log(Object.getOwnPropertyDescriptor(produto, 'nome'));
console.log(Object.values(produto));

for(let entry of Object.entries(produto)){
    console.log(entry);
}
console.log(Object.entries(produto));

/* Aula 89 - Prototypes

/*
    #Nota#

    _ JavaScript é baseado em protótipos para passar propriedades e método
s
    _ de um objeto para outro.

```

```

_ Definição de protótipo
_ Protótipo é o termo usado para se referir ao que foi criado pela
_ primeira vez, servindo de modelo ou molde para futuras produções.

_ Todos os objetos tem uma referência interna para um protótipo (__proto__)
_ que vem da propriedade prototype da função construtora que foi usada
para criá-lo.
_ Quando tentamos acessar um membro de um objeto, primeiro o motor do
JS
_ vai tentar encontrar este membro no próprio objeto e depois a cadeia
de protótipos
_ é usada até o topo (null) até encontrar (ou não) tal membro;

#Notas Aula#
_ todo objeto criado pelo molde será associado a um prototype -> proto
! Uma afirmação que deve ser lembrada é:
! pessoa.prototype === pessoa1.__proto__
*/

function Pessoa(nome, sobrenome){
  this.nome = nome;
  this.sobrenome = sobrenome;
  this.nomeCompleto = () => this.nome + ' ' + this.sobrenome;
}
pessoa1.prototype.estaAqui = 'Hahahaha';

Pessoa.prototype.nomeCompleto = function(){
  return this.nome + ' ' + this.sobrenome;
};

const pessoa1 = new Pessoa('Jean', 'Meira')
console.log(pessoa1);

/* Aula 90 - Manipulando Prototypes
*/
#Nota#
_ A manipulação dos prototypes permite criar todos os métodos gerados
por
_ constructor function centralizados em um único lugar, isso poupa recursos
_ de máquina e é mais performático.

```

- \_ é possível setar um prototype em um objeto, o que torna o código ainda mais reutilizável.
- \_ O sendo possível assim trabalhar com métodos de um objeto que vem de uma constructor em
- \_ outros objetos.

```
*/
```

```
/// exemplo 1
```

```
const objA = {
  chaveA: 'A'
}
const objB = {
  chaveB: 'B'
}
const objC = {
  chaveC: 'C'
}
```

```
Object.setPrototypeOf(objB, objA); //? -> muda a cadeia, mas no fim ainda
chega em Object.prototype;
console.log(objB.chaveA);
```

```
Object.setPrototypeOf(objC, objB);
console.log(objC.chaveA, objC.chaveB);
```

```
/// exemplo 2
```

```
function Produto(nome, preco){
  this.nome = nome;
  this.preco = preco;
}
```

```
Produto.prototype.desconto = function(percentualDeDesconto){
  this.preco = this.preco*(1 - (percentualDeDesconto/100));
  this.preco = this.preco.toFixed(2);
}
```

```
Produto.prototype.aumento = function(percentualDeAumento){
  this.preco = this.preco * (1 + (percentualDeAumento/100));
  this.preco = this.preco.toFixed(2);
}
```

```
const produto1 = new Produto('camiseta', 100);
const produto2 = new Produto('camiseta2', 100);
console.log(produto1);
```

```

produto1.desconto(10);
console.log(produto1);

console.log(produto2);
produto2.aumento(10);
console.log(produto2);

const produto3 = {
  nome: 'caneca',
  preco: 15
};
Object.setPrototypeOf(produto3, Produto.prototype);
console.log(produto3);
produto3.aumento(10);
console.log(produto3);

const produto4 = Object.create(Produto.prototype, {
  preco: {
    writable: true,
    configurable: true,
    enumerable: true,
    value: 42,
  },
  tamanho: {
    writable: true,
    configurable: true,
    enumerable: true,
    value: 40,
  }
});
produto4.aumento(10);
console.log(produto4);

```

### /\* Aula 91 - Herança

```

/*
#Nota#
_ A herança permite abstrair e depois especificar.
_ dessa forma se isola partes comuns do código, mas também atribuir
_ partes específicas com isso, é possível pegar objetos diferentes
_ como de camisetas e canecas, e associar ambos como produtos,
_ que conterão nome e preço em comum, além dos métodos de aumentar ou
_ diminuir os preços. Mas a camiseta tem cor e a caneca tem material,
_ que o outro objeto não tem. Assim, permitesse associar ambos a um

```

```

    _ construtor Produto(), abstraindo o nome e preço, e depois especifica
r
    _ com adicionais no construtor Camiseta() e Caneca(), dentro do constr
utor
    _ específico se usa o Produto.call(this, nome, preco); referenciando o
    _ objeto atual com o construtor. Produto e passando os argumentos para
    _ nome e preco nesse caso (que são necessários em Produto), o call sim
ula
    _ o new em criar um objeto vazio e fazer o this apontar pra ele, call
é um método
    _ presente em toda função.
*/

//produto -> aumento e desconto
// camiseta = cor , caneca = material

function Produto(nome, preco){
    this.nome = nome;
    this.preco = preco;
}

Produto.prototype.aumento = function(quantia){
    this.preco += quantia;
};

Produto.prototype.desconto = function(quantia){
    this.preco -= quantia;
};

function Camiseta(nome, preco, cor){
    Produto.call(this, nome, preco);
    this.cor = cor;
}

Camiseta.prototype = Object.create(Produto.prototype);
//? associa os métodos e afins
Camiseta.prototype.constructor = Camiseta;
//? corrige o problema de mudança no construtor de origem

function Caneca(nome, preco, material){
    Produto.call(this, nome, preco);
    this.material = material;
}

Caneca.prototype = Object.create(Produto.prototype);
Caneca.prototype.constructor = Caneca;

```

```

const camiseta = new Camiseta('regata', 7.5, 'preta');
console.log(camiseta);
camiseta.aumento(10);
console.log(camiseta);

const caneca = new Caneca('caneca de café', 10.0, 'porcelana');
console.log(caneca);
caneca.aumento(10);
console.log(caneca);

/*
#Nota explicativa passada pelo professor#
_ O termo "Herança" é muito usado para descrever que as
_ características de um objeto são passadas para outro objeto.
_ Assim como uma filha ou filho pode herdar características da
_ sua mãe ou pai (geneticamente). Por isso estamos usando esse
_ termo para descrever o comportamento desejado.
_ Além disso, a maioria das literaturas, incluindo as documentações
_ (como MDN) citam a palavra herança.

_ Vamos continuar usando o termo "Herança" até o final do curso,
_ porém, Javascript não é uma linguagem que apresenta essa característ
ica
_ em específico, portanto não existe "herança" aqui. O que ocorre,
_ na verdade, é "delegação". Esse seria o termo mais correto para
_ descrever "Herança" em JS.

_ Mais especificamente no trecho:

_ Camiseta.prototype = Object.create(Produto.prototype);
_ Estamos, claramente, falando que o Camiseta.prototype é um novo obje
to
_ vazio que tem como protótipo Produto.prototype. Dessa forma, estamos
adicionando
_ Camiseta.prototype na cadeia de protótipos de Produto. Por consequên
cia,
_ tudo o que não existir em
! Camiseta nem em Camiseta.prototype será delegado para Produto.protot
ype
_ (e assim por diante até chegar no topo da cadeia de protótipos).
_ Por isso vejo o termo delegação como mais adequado para este comport
amento.
_ Novamente, vamos continuar usando o termo "Herança", mas achei que
_ seria interessante que você soubesse disso neste momento.
*/

```



## /\*\* Aula 94 - Resolução válida CPF

// 705.484.450-52 070.987.720-03

/\*

7x 0x 5x 4x 8x 4x 4x 5x 0x

10 9 8 7 6 5 4 3 2

70 0 40 28 48 20 16 15 0 = 237

11 - (237 % 11) = 5 (Primeiro dígito)

Se o número digito for maior que 9, consideramos 0.

7x 0x 5x 4x 8x 4x 4x 5x 0x 5x

11 10 9 8 7 6 5 4 3 2

77 0 45 32 56 24 20 20 0 10 = 284

11 - (284 % 11) = 2 (Primeiro dígito)

Se o número digito for maior que 9, consideramos 0.

\*/

```
function ValidaCPF(cpfEnviado) {
  Object.defineProperty(this, 'cpfLimpo', {
    enumerable: true,
    get: function() {
      return cpfEnviado.replace(/\D+/g, '');
    }
  });
}
```

```
ValidaCPF.prototype.valida = function() {
  if(typeof this.cpfLimpo === 'undefined') return false;
  if(this.cpfLimpo.length !== 11) return false;
  if(this.isSequencia()) return false;

  const cpfParcial = this.cpfLimpo.slice(0, -2);
  const digito1 = this.criaDigito(cpfParcial);
  const digito2 = this.criaDigito(cpfParcial + digito1);

  const novoCpf = cpfParcial + digito1 + digito2;
  return novoCpf === this.cpfLimpo;
};
```

```
ValidaCPF.prototype.criaDigito = function(cpfParcial) {
  const cpfArray = Array.from(cpfParcial);

  let regressivo = cpfArray.length + 1;
  const total = cpfArray.reduce((ac, val) => {
```

```

    ac += (regressivo * Number(val));
    regressivo--;
    return ac;
}, 0);

const digito = 11 - (total % 11);
return digito > 9 ? '0' : String(digito);
};

ValidaCPF.prototype.isSequencia = function() {
    const sequencia = this.cpfLimpo[0].repeat(this.cpfLimpo.length);
    return sequencia === this.cpfLimpo;
};

const cpf = new ValidaCPF('070.987.720-03');

if(cpf.valida()) {
    console.log('Cpf válido');
} else {
    console.log('Cpf inválido');
}

/** Aula 95 - Polimorfismo

/*
    #Nota#
    _ Como fazer métodos se comportarem de maneiras diferentes.
    _ como subclasses -> que são classes filhas
    _ por classes entende-se -> função construtora;
    _ A Superclasse / classe pai ou mãe é o que dará origem as demais, que
    serão
    _ filhas ou sub-classes. Os métodos da superclasse estarão nas sub-
    classes,
    _ mas ao usar o polimorfismo é possível adotar comportamentos diferentes para
    _ o mesmo método, somente nas filhas que forem desejados ter comportam
    entos
    _ diferenciados.
    _ Ao final do código é possível notar que ambas as classes filhas herd
    am
    _ características da classe mãe, mas a conta corrente tem o método de
    sacar
    _ sobreposto, portanto tem esse comportamento em específico diferente
    de sua classe mãe.
*/

```

```

//#Nota# Superclasse / classe pai ou mãe

function Conta(agencia, conta, saldo){
    this.agencia = agencia;
    this.conta = conta;
    this.saldo = saldo;
}

//_ Métodos da superclasse.
Conta.prototype.sacar = function(valor){
    if(valor > this.saldo) {
        return console.log(`Saldo insuficiente para sacar R${valor}, saldo: R${this.saldo}`);
    }
    this.saldo -= valor;
    this.verSaldo();
};

Conta.prototype.depositar = function(valor){
    this.saldo += valor;
    this.verSaldo();
};

Conta.prototype.verSaldo = function(){
    console.log(`Ag./Conta: ${this.agencia}/${this.conta} - saldo: R${this.saldo.toFixed(2)}`);
    return this.saldo;
};

//_ manipulação de um novo objeto criado com a super classe.
const conta1 = new Conta(11, 22, 10);
console.log(conta1);
conta1.depositar(11);

conta1.depositar(10);

conta1.sacar(30);
conta1.sacar(10);

//_ Especialização para a primeira classe filha, conta corrente.
function ContaCorrente(agencia, conta, saldo, limite){
    Conta.call(this, agencia, conta, saldo);
    //? Linkando com a superclasse.
    this.limite = limite;
}

```

```

}
//_ Linkando a cadeia de prototypes
ContaCorrente.prototype = Object.create(Conta.prototype);
//_ mas mantendo a referência de quem é o construtor.
ContaCorrente.prototype.constructor = ContaCorrente;

//! Sobrepondo o método sacar, para se ter comportamento diferente da
superclasse.
ContaCorrente.prototype.sacar = function(valor){
    if(valor > (this.saldo + this.limite)) {
        return console.log(`Saldo insuficiente para sacar R${valor}, sald
o: R${this.saldo}`);
    }
    this.saldo -= valor;
    this.verSaldo();
};

//_ Manipulando a classe filha conta corrente.
const CC1 = new ContaCorrente(33, 44, 0, 100);
CC1.depositar(10);
CC1.sacar(110);

//_ Especialização para a segunda classe filha, conta poupança.
function ContaPoupanca(agencia, conta, saldo, limite){
    Conta.call(this, agencia, conta, saldo);
    //? Linkando com a superclasse.
    this.limite = limite;
}

//_ Novamente Linkando a cadeia de prototypes
ContaPoupanca.prototype = Object.create(Conta.prototype);
//_ mas novamente mantendo a referência de quem é o construtor.
ContaPoupanca.prototype.constructor = ContaPoupanca;

//_ Manipulando a classe filha conta corrente.
const CP1 = new ContaPoupanca(55, 66, 0);
CP1.depositar(10);
CP1.sacar(110);

/* Aula 96 - Factory Function + Prototypes.
/*
#Nota#

```

\_ O retorno pode estar linkando o prototype já dentro dela, ou você pode desacoplar

\_ em partes distintas e ter um objeto que contenha os métodos em uma variável.

\_ então essa variável pode ser passada para o prototype do objeto criado.

\*/

```
const falar = {
  falar() {
    console.log(`${this.nome} está falando`);
  },
}
const comer = {
  comer() {
    console.log(`${this.nome} está comendo`);
  },
}
const beber = {
  beber() {
    console.log(`${this.nome} está bebendo`);
  }
}

const pessoaPrototype = {...falar, ...comer, ...beber }

function criaPessoa(nome, sobrenome) {
  // const pessoaPrototype = {
  //   falar(){
  //     console.log(`${this.nome} está falando`);
  //   },
  //   comer(){
  //     console.log(`${this.nome} está comendo`);
  //   },
  //   beber(){
  //     console.log(`${this.nome} está bebendo`);
  //   },
  // };
  return Object.create(pessoaPrototype, {
    nome: { value: nome },
    sobrenome: { value: sobrenome }
  });
}

const p1 = criaPessoa('Jean', 'Meira');
console.log(p1.nome, p1.sobrenome, p1);
```

```

p1.falar();

/** Aula 97 - Objeto Map()

/*
    #Nota#
    _ O algoritmo que coloca os valores em novasPessoas com o laço for of
    _ tem dois problemas, o primeiro é que o id passado para as posiçõ
es
    _ será convertido de números para strings e o segundo é que graças a u
m
    _ algoritmo interno do javascript a ordem será crescente a partir dos
ids.
    _ mesmo que a converção final trate como string, a ordenação geral ser
á
    _ como se fossem números em ordem crescente.
    _ exemplo.
    _ {
    _ '1': { id: 1, nome: 'Jose' },
    _ '2': { id: 2, nome: 'Maria' },
    _ '3': { id: 3, nome: 'Luiz' }
    _ }
    _ vemos os ids como '1', '2' e '3', mostrados como strings.
    _ e a ordenação ficou em ordem crescente.
    ! Pode-se criar como um Map.
    _ Então a estrutura de par chave e valor continua a mesma.
    _ mas com alterações, o id continua a ser um número
    _ e a ordenação ficou a mesma da original.
    _ e há mudanças pequenas, para colocar um valor se usa o set
    _ para ler uma valor é usado o get.
    _ é possível iterar normalmente no map.

*/

const pessoas = [
    { id: 3, nome: 'Luiz' },
    { id: 2, nome: 'Maria' },
    { id: 1, nome: 'Jose' },
];

const novasPessoas = {};
for (const { id, nome } of pessoas) {
    novasPessoas[id] = { id, nome }
}

```

```

console.log(novasPessoas);

const novasPessoas2 = new Map();
for (const { id, nome } of pessoas) {
    novasPessoas2.set(id, { id, nome });
}
console.log(novasPessoas2);
console.log(novasPessoas2.get(2));

//| iterando no map

for(const [identifier, {id, nome}] of novasPessoas2){
    console.log(identifier, id, nome);
}

for(const pessoas of novasPessoas2){
    console.log(pessoas);
}

//Classes

```

/\* Aula 98 - Criando Classes.

```

/*
    #Nota#
    _ A criação remete ao mesmo padrão de uma constructor function, mas usa o class
    _ Criação de métodos não precisam de ; e também são adicionados diretamente ao
    _ prototype da classe em questão.
    _ A diferença é que o constructor será uma classe.
*/

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

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

```

```
    beber(){  
        console.log(`${this.nome} está bebendo`);  
    }  
}  
const p1 = new Pessoa('Jean', 'Meira');  
  
console.log(p1);
```