Vetores, Matrizes e Laço for 11/11/20

Um array é como uma variável, no entanto, ele é capaz de armazenar vários valores, não apenas um.

Também é chamado em português de vetor ou arranjo e no caso do javascript não seria errado chamá-lo de lista.

A sintaxe de um array é a seguinte:

const array = [5, 4, 3.5, 4, "banana", null, undefined, "chuva"];

Entre colchetes, podemos declarar uma lista de valores de qualquer tipo, em qualquer ordem, separados por vírgula. Os valores são organizados pela ordem em que aparecem no array.

Para acessarmos esses valores o array se vale de um sistema de endereçamento, cada valor tem um endereço no array que chamamos de index ou índice. Na maioria das vezes o índice será a posição em que o valore se encontra no array.

**Mas atenção!** Os índices de um array sempre começam do **0**.

Isso significa que um array de 5 elementos tem índices 0, 1, 2, 3 e 4.

Para acessar um valor não basta mais usarmos o nome da variável onde está o array, precisamos dizer também qual índice queremos olhar.

Para isso, passamos entre colchetes, após o nome da variável o índice desejado.

Usando o exemplo acima como array:

array[0]; // 5

array[3]; // 4

array[4]; // "banana"

É importante notar que, para um array de n elementos, o índice deve estar entre 0 e n-1. Usar um índice fora desse intervalo provocará um erro.

Podemos usar a mesma sintaxe para fazer uma atribuição e assim alterar o valor de uma determinada posição do array:

array[0] = 3;

No javascript, arrays são objetos e tem propriedades e métodos. Métodos são funções que geralmente manipulam os valores internos do array. Temos uma grande quantidade dessas funções nos arrays, que permitem adicionar items, remover item, procurar por valores e etc...

Tamanho de um array

Antes de vermos os métodos vamos ver a única propriedade indispensável de conhecer dos arrays.

Essa propriedade chama-se length e nos dá o tamanho do array:

*Array.length()*

*const arr = [1,2,3,4];*

*console.log(arr.length); // 4*

Vale mencionar que se você quiser saber o último índice de um array basta fazer arr.length -1.

Inserção de valores ao array

Vamos começar com um array contendo o valor 11 e adicionar o valor 10 ao final usando push();

*Array.push()*

*const arr = [11];*

*arr.push(10);*

*O elemento 10 é inserido no index 1 de arr.*

*push() retorna o novo tamanho do array se precisar dele basta atribuir a uma variável.*

*const size = arr.push(10)*

Se quisermos adicionar no começo do array, basta usar unshift() no lugar de push()

*Array.unshift()*

*const arr = [11];*

*arr.unshift(10);*

*O elemento 10 é inserido no index 0 de arr.*

*unshift() como push() também retorna o novo tamanho do array.*

Busca de valores em um array

Como encontrar um valor dentro do array? Com indexOf ou lastIndexOf().

Vamos criar um array com duas vezes o número 10 e procurar onde ele está:

*Array.indexOf e Array.lastIndexOf()*

*const arr = [10, 10];*

*const firstIndex = arr.indexOf(10); // 0*

*const lastIndex = arr.lastIndexOf(10); // 1*

*indexOf retorna a primeira posição (index) onde o valor foi encontrado, enquanto lastIndexOf retorna a última.*

*Se o valor não está presente no array esses métodos retornam -1, assim podemos usá-los também para saber se um determinado valor existe no array.*

*const arr = [10, 10];*

*console.log(arr.indexOf(10) > -1); // true*

*console.log(arr.indexOf(11) > -1); // false*

No entanto, é mais fácil usar o método includes() que já retorna um booleano.

*Array.includes()*

*const arr = [10, 10];*

*console.log(arr.includes(10)); // true*

*console.log(arr.includes(11)); // false*

includes() permite um parâmetro opcional se você desejar procurar um valor a partir de uma posição que não seja o início do array.

Remoção de valores em um array

Temos 2 opções para remover items de um array, pop(), shift().

Array.pop()

const arr = [1, 2, 3, 4, 5];

const lastValue = arr.pop();

console.log(arr) // [1, 2, 3, 4]

console.log(lastValue) // 5

pop() remove o último item do array e o retorna.

Array.shift()

const arr = [1, 2, 3, 4, 5];

const firstValue = arr.shift();

console.log(arr) // [2, 3, 4, 5]

console.log(firstValue) // 1

shift() remove o primeiro item do array e o retorna.

Splice

Array.splice() é um dos métodos mais confusos de um array, ele pode remover e adicionar itens ao mesmo tempo. Ele tem um número variável de parâmetros o que deixa as coisas ainda mais confusas... Mas vamos encarrar...

*Array.splice() com 1 parâmetro.*

*const arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];*

*const res = arr.splice(3);*

*console.log(res); // [4, 5, 6, 7, 8, 9, 10]*

*console.log(arr); // [1, 2, 3]*

Veja, o splice retornou um array contendo os elementos do array original a partir do index 3, ou seja, a partir do valor 4, que está no index 3.

Mas ele também modificou o array original, deixando nele apenas a parte não selecionada, do index 0 até o index 2. Sendo assim, nele ficaram os valores 1, 2 e 3.

*Array.splice() com 2 parâmetros.*

*const arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];*

*const res = arr.splice(3,2);*

*console.log(res); // [4, 5]*

*console.log(arr); // [1, 2, 3, 6, 7, 8, 9, 10]*

*Veja, o splice retornou um array contendo os elementos do array original a partir do index 3, ou seja, a partir do valor 4, que está no index 3. No entanto, como passamos o valor 2 como segundo parâmetro ele pegou apenas 2 elementos e manteve todos os demais no array original.*

Sendo assim, o array original ficou com 1, 2 e 3, valores anteriores ao index 3, e com 6, 7, 8, 9, 10 valores posteriores aos dois itens removidos (index 3 e 4).

Em suma, com dois parâmetros, o splice tira uma fatia do array a partir de um index com o tamanho indicado pelo segundo parâmetro.

Para deixar as coisas ainda mais complicadas, o splice aceita números negativos no primeiro parâmetro, o que faz com que ele comece pelo final do array. Isso é útil para remover e retornar, por exemplo, os dois últimos valores arr.splice(-2,2).

Array.splice() com N parâmetros.

*const arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];*

*const res = arr.splice(3,2,11,12,13);*

*console.log(res); // [4, 5]*

*console.log(arr); // [1, 2, 3, 11, 12, 13, 6, 7, 8, 9, 10]*

Se após o segundo parâmetro forem passados mais parâmetros ao splice ele adicionará todos os valores passados no array resultante. No entanto, ele adicionará onde recortamos o array original. No exemplo acima, os valores 11, 12 e 13 foram adicionados onde os valores removidos estavam originalmente (index 3).

Você pode passar quantos parâmetros quiser.

Preenchendo um Array

Podemos usar o método fill() para preencher um array com um valor padrão. No entanto, para que ele funcione, precisamos criar o array de uma forma diferente.

Dissemos que arrays são objetos, sendo assim, eles possuem construtores. O construtor do array permite que passemos um tamanho inicial ao array. O método fill() precisa desse tamanho para saber quantos itens adicionar.

*Array.fill()*

*const arr = new Array(10);*

*arr.fill(0);*

*console.log(arr); // [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]*

Concatenando Arrays

Assim como strings, dois arrays podem ser concatenados para formar um único array com os elementos de ambos.

*Array.concat()*

*const arr1 = [1, 2, 3, 4];*

*const arr2 = [5, 6, 7, 8];*

*const arr3 = arr1.concat(arr2);*

*console.log(arr3); // [1, 2, 3, 4, 5, 6, 7, 8]*

A melhor parte de usar concat é que ele não modifica nenhum dos arrays originais, ele apenas cria um novo com todos os elementos...

Como concat produz um array como retorno podemos encadear várias chamadas dele:

*Array.concat() encadeados*

*const arr1 = [1, 2, 3, 4];*

*const arr2 = [5, 6, 7, 8];*

*const arr3 = [9, 10];*

*const arr4 = arr1.concat(arr2).concat(arr3);*

*console.log(arr4); // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]*

Revertendo Arrays

Se desejarmos inverter todos os valores de um array basta usar o método reverse().

*Array.reverse()*

*const arr = [1, 2, 3, 4];*

*arr.reverse();*

*console.log(arr); // [4, 3, 2, 1]*

Lembre-se que reverse() altera o array original!

Ordenação

Arrays podem ser ordenados pelo método sort() no entanto vale ressaltar que o javascript só é capaz de ordenação numérica ou alfabética (Number ou String).

Ordenações mais complexas requerem que você crie uma função de ordenação...

const arr = [5,7,3,4,2,1];

arr.sort()

console.log(arr); // [1, 2, 3, 4, 5, 7]

const arr2 = ["carla", "alberto", "bruna"];

arr2.sort();

console.log(arr2); // ["alberto", "bruna", "carla"]

Lembre-se que sort() altera o array original!

Outros métodos omitidos aqui

Arrays tem muitos outros métodos, escolhemos omiti-los por dois motivos, alguns são funções de alta ordem, e veremos como usá-las mais a frente, outros são pouco utilizados e você pode estudá-los se tiver a necessidade.

Use o [MDN](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array) como referência em seus estudos.

Cuidado com os métodos que tem um erlenmeyer na frente, eles podem não estar disponíveis em todos navegadores! Também os que tem um lixinho na frente já foram depreciados e não devem ser usados.

Sobre const e arrays

Veja que em todos os exemplos usamos const e ainda assim pudemos modificar os valores do arrays, inserir, remover e etc...

Array, como dissemos, é um objeto e const não protegem os objetos de manipulação interna apenas de reatribuição, ou seja, é qualquer alteração no array funcionará, no entanto, tentar colocar um novo array na mesma constante é impossível.

const arr = [10];

arr = [10, 20]; // erro

const arr = [10];

arr.push(20); // [10, 20]

Tenha cuidado em manipular seus arrays para não perder os valores originais, prefira formas de preservar o conteúdo original e produzir novos arrays modificados. Vamos ver mais para frente métodos funcionais que fazem exatamente isso, assim como sintaxes especiais do ES6 que são capazes de fazer tudo o que todos os métodos que vimos fazem sem modificar os arrays.

# Laço de Repetição for

Temos 3 formas do laço de repetição for, duas delas são para uso com arrays, por isso guardamos ele para ser mostrado aqui e não na lição de laços.

### **for-in**

*for-in percorre todos os index de um array*

*const arr = [10,20,30,40,50];*

*for(index in arr){*

*console.log(index, arr[index]); // mostra o index e o valor dessa rodada*

*}*

*/\* saída*

*0 10*

*1 20*

*2 30*

*3 40*

*4 50*

*\*/*

### **for-of**

*for-of percorre todos os valores de um array*

*const arr = [10,20,30,40,50];*

*for(value of arr){*

*console.log(value); // mostra o valor dessa rodada*

*}*

*/\* saída*

*10*

*20*

*30*

*40*

*50*

*\*/*

Muito cuidado ao alterar os items de um array dentro de um laço de repetição.

Se o array mudar de tamanho durante o laço, ele poderá se comportar de forma imprevisível ou gerar erros. Evite adicionar ou remover items do mesmo array que você está percorrendo, utilize técnicas que criam novos arrays no lugar.

## Laço for em sua forma genérica

O laço for é um laço com critério de parada numérica, isso quer dizer que ele é indicado para problemas em que a repetição deve ser feita um número determinado de vezes.

O comando for tem 3 partes, o valor inicial, o valor final (booleano) e um incremento, normalmente chamado de passo (step), que é a modificação do valor atual em direção ao valor final.

Isso quer dizer que podemos fazer um for que conta do número 1 ao 10 de 1 em 1.

Assim como podemos fazer um for que começa em 2 e vai até 10 de 2 em 2 para passar apenas pelos pares.

for utiliza uma variável interna para registrar em que momento do laço ele está, devido ao seu uso em conjunto com arrays normalmente chamamos essa variável de i simbolizando index.

*for(let i = 0; i < 10; i++){ // de 0 até 9, de 1 em 1*

*console.log('Olá, pela ' + i + 'ª vez...');*

*}*

Podemos usar o for comum para percorrer um array apesar de for-of ou for-in serem muito mais práticos:

*const arr = [10, 20, 30];*

*for(let i = 0; i < arr.length; i++){ // de 0 até 2, de 1 em 1*

*console.log(arr[1]);*

*}*

*/\* Saída*

*10*

*20*

*30*

*\*/*

## for, while ou do-while

Quando usar cada laço?

Minha sugestão é a seguinte, quando você controlar um laço numéricamente prefira o for essa é a maior parte dos casos (raramente uso while), no entanto, quando a decisão é mais complexa ou não numérica pode valer a pena recorrer ao while.

Também precisamos optar pelo while quando não sabemos quando parar, por exemplo, continue até que a variável x seja null é o tipo de laço em que é muito mais prático usando while.

Entre while e do-while nós já conversamos, basta escolher se você quer que a condição de parada seja no começo ou no final da rodada do laço.

Inclusive, o nome correto para uma rodada é uma iteração.

## Mais um exemplo...

Você deve estar pensando que esqueci a promessa de fazer um novo programa de rolagem de dados usando essas coisas novas que aprendemos, certo?

Não esqueci, vamos lá!

Temos novos requisitos para esse programa.

No D&D, é muito comum, especialmente para os magos, usar vários dados do mesmo tipo.

Por exemplo, a magia "magic missile" no nível mais alto dá 5 mísseis que fazem 1 dado de 4 lados + 1 de dano.

Sendo assim, o jogador rola 5 dados de 4 lados e soma 5.

No caso da fireball, magia favorita dos magos com tendências incendiárias, pode chegar até 10 dados de 6 lados de dano! E ainda pode queimar seu equipamento e derreter suas moedas =[

Para simplificar, no D&D falamos 5D4+5 ou 10D6 para essas rolagens. Será isso que receberemos do usuário agora, quantos dados e de quantos lados.

Primeiro, vamos transformar "5D4" em [5, 4] usando o método split de string que produz um array.

Depois faremos todas as rolagens e informaremos ao usuário em duas formas, os valores de cada rolagem e a soma (que geralmente é o que ele deseja...).

Vamos dividir o código, primeiro as funções que já tínhamos:

*function randomIntFromInterval(min, max) {*

*return Math.floor(Math.random() \* (max - min + 1) + min);*

*}*

*function roll(dice) {*

*return randomIntFromInterval(1, dice);*

*}*

Adicionamos uma função para lidar com o input:

*function splitInput(input){*

*const splitedInput = input.toUpperCase().split('D');*

*const normalizedInput = [parseInt(splitedInput[0]), parseInt(splitedInput[1])];*

*return normalizedInput;*

*}*

O que ela faz é pegar a entrada do usuário, alterar para todas as letras maiúsculas usando toUpperCase() (assim o usuário pode digitar o D maiúsculo ou minúsculo) e usa o split() que quebra uma string em um determinado caractere, no caso "D", e retorna um array com as partes.

Para normalizar os dados que estão em forma de string, criamos um novo array com a versão numérica de cada parte usando parseInt.

Assim passamos de "5D10" para ["5","10"] e finalmente para [5,10], que é o que precisamos.

Vamos ao programa:

*const input = prompt('digite a quantidade e o dado no formato D&D, ex. 1D4');*

*const quantityAndDice = splitInput(input);*

*const quantity = quantityAndDice[0];*

*const dice = quantityAndDice[1];*

*let rolls = [];*

*let total = 0;*

*for(let i = 0; i < quantity; i++){*

*const currentRoll = roll(dice);*

*rolls.push(currentRoll);*

*total += currentRoll;*

*}*

*console.log("Rolagens: " + rolls + " Soma: " + total);*

Explicando linha a linha:

Solicitamos ao usuário digitar a quantidade e o dado separados por D.

Aplicamos a função que produz um array com a quantidade e o tipo do dado.

Nas duas próximas linhas, separamos esse array em duas constantes por comodidade.

Declaramos rolls como array vazio.

Declaramos total como 0.

No laço, faremos N rolagens, sendo N a quantidade que o usuário solicitou.

Usamos a função roll() para fazer a rolagem.

Adicionamos a rolagem ao array rolls.

Incrementamos o valor de total com o valor da rolagem.

Terminamos o laço e podemos mostrar os resultados!

### **E o laço??**

Optamos por remover o laço externo que faz esse programa rodar até que o usuário faça um input vazio. Mas podemos adicioná-lo. Nesse exemplo, as entradas são mais complexas e há mais margem para erro do que no da lição passada, então teríamos que fazer uma validação da entrada muito bem feita.

Vamos mostrar um exemplo usando uma expressão regular, que é a melhor forma de resolver esse problema, mas isso está fora do escopo dessa lição, será meramente ilustrativo:

Primeiro alteramos a função que faz o split do input para usar uma expressão regular e produzir 3 possibilidades de retorno, undefined se o usuário não digitou nada, null se digitou algo fora do padrão esperado e array se ele fez tudo certo.

*function splitInput(input) {*

*if (!input) return undefined;*

*const regex = /(\d+)D(4|6|8|10|12|20)/;*

*const result = input.toUpperCase().match(regex);*

*if (result && result[0] === input.toUpperCase()) {*

*return [parseInt(result[1]), parseInt(result[2])];*

*}*

*else return null;*

*}*

Na regex dissemos que qualquer quantidade numérica de pelo menos 1 dígito pode ser usada (\d+), seguida de "D" (por que fizemos toUpperCase()) e seguida de um dos dados, podendo ser 4, 6, 8, 10, 12 ou 20.

Usamos parênteses para fazer grupos de captura, um para quantidade e um para o dado, assim o método match() já nos dá um array com tudo o que precisamos.

Na primeira posição de result deve estar a entrada do usuário se bem formatada, em seguida a quantidade e por último o dado. ex. "10D4" produz ["10D4", "10", "4"].

Entradas mal sucedidas produzem null no result, e entradas parcialmente bem sucedidas produzem algo diferente da entrada do usuário na primeira posição, por isso testamos ambos, if (result && result[0] === input.toUpperCase()).

Se tudo deu certo estamos prontos, fazemos o parseInt() normalmente para transformar os valores em números e retornamos (veremos return logo mais em funções).

No final, o programa precisa lidar com as três opções de retorno dessa função, se for undefined ele para o laço, se for null ele mostra uma mensagem na tela dizendo que a entrada está incorreta e no último caso faz as rolagens e tudo mais.

*for(;;) {*

*const input = prompt('digite a quantidade e o dado no formato D&D, ex. 1D4');*

*const quantityAndDice = splitInput(input);*

*if (quantityAndDice === undefined) break;*

*else if (quantityAndDice === null) console.log('entrada em formato inválido');*

*else {*

*const quantity = quantityAndDice[0];*

*const dice = quantityAndDice[1];*

*let rolls = [];*

*let total = 0;*

*for (let i = 0; i < quantity; i++) {*

*const currentRoll = roll(dice);*

*rolls.push(currentRoll);*

*total += currentRoll;*

*}*

*console.log("Rolagens: " + rolls + " Soma: " + total);*

*}*

*}*

Aproveitei para trocar o do-while por um for infinito for(;;) já que estamos vendo laços for...

O comando break serve para sair arbitráriamente do laço, ele pode ser usado com qualquer laço de repetição. Um cuidado que temos que ter é que quando temos um laço dentro de outro o break só sairá de laço onde ele está dentro. Da mesma forma, se há um switch dentro de um laço não precisamos nos preocupar que o break do switch encerre o laço.

Vamos agora dar uma olhada em matrizes...

# Matrizes

Arrays são valores como qualquer outro, nada impede que tenhamos arrays dentro de outro array. Isso seria um array bidimensional ou uma matriz.

Você pode pensar em uma matriz como sendo uma tabela, para endereçar uma determinada célula precisamos saber o número da linha e da coluna.

Como essa tabela será construída com arrays aninhados (um dentro do outro), precisamos lembrar que vale a regra dos índices começando em zero!

const tabela = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 9]

];

No exemplo, o array maior, que foi alocado na variável, representa as linhas da tabela, enquanto os arrays internos representam as colunas.

Sendo assim tabela[0] representa a primeira linha, ou se preferir, o array [1,2,3].

console.log(tabela[0]); // [1, 2, 3]

console.log(tabela[1]); // [4, 5, 6]

console.log(tabela[2]); // [7, 8, 9]

Viu? Acessar as linhas foi fácil, agora vamos acessar as células individualmente. Se tabela[0] é um array, e acessamos um valor no array fazendo arr[index], podemos acessar os valores da célula da mesma forma: tabela[0][index], sim dois pares de colchetes, pois são dois arrays, um dentro do outro...

console.log(tabela[0][0]); // 1

console.log(tabela[0][1]); // 2

console.log(tabela[0][2]); // 3

Vamos percorrer a tabela toda manualmente para ver um padrão que emerge de nosso código:

console.log(tabela[0][0]); // 1

console.log(tabela[0][1]); // 2

console.log(tabela[0][2]); // 3

console.log(tabela[1][0]); // 4

console.log(tabela[1][1]); // 5

console.log(tabela[1][2]); // 6

console.log(tabela[2][0]); // 7

console.log(tabela[2][1]); // 8

console.log(tabela[2][2]); // 9

Observe o padrão, travamos a linha em 0, depois passamos por todos os itens da linha variando o index da coluna de 0 até 2. Depois passamos para a próxima linha (1) e fazemos a mesma coisa. Por fim, chegamos a última linha (2) e fazemos a mesma coisa...

Fazer a mesma coisa é especialidade dos laços de repetição!

Devemos poder usá-los para simplificar esse código, imagine como ele seria de fosse uma matriz 100x100...

Vamos começar com apenas uma laço nas linhas:

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

console.log(tabela[i][0]);

console.log(tabela[i][1]);

console.log(tabela[i][2]);

}

/\* Saída

1

2

3

4

5

6

7

8

9

\*/

Veja que o que sobrou de código é o acesso da coluna 0 até 2, podemos fazer outro laço que faz isso:

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

for(let j = 0; j < tabela[i].length; j++){

console.log(tabela[i][j]);

}

}

/\* Saída

1

2

3

4

5

6

7

8

9

\*/

Lembra das matrizes da escola? Lá endereçávamos os valores por i e j, sendo um a linha e o outro a coluna, da mesma forma usávamos colchetes para fazer essa indexação, tudo isso é igual na programação.

Não sei você, mas eu odiava matrizes na escola...

Isso acontece porque elas estavam fora de um contexto significativo, matrizes não traziam nenhum benefício para minha vida na escola. Diferentemente, na programação elas nos auxiliam bastante e são uma estrutura de dados muito importante.

No exemplo acima, cuidado com a sopa de letrinhas, veja que no segundo for uso tabela[i].length para pegar a quantidade de colunas, é muito fácil trocar um i por um j e gerar caos.

Esse exemplo tem uma grande vantagem, na programação usamos arrays aninhados e não matrizes, o que quer dizer que não temos algumas limitações, por exemplo, ninguém garante que o número de colunas seja o mesmo para cada linha...

No exemplo, obtemos o tamanho da coluna dinamicamente usando length, o que funcionará mesmo em exemplos com colunas variáveis.

Veja:

const tabela = [

[1, 2, 3],

[4, 5, 6, 7],

[8, 9, 10, 11, 12]

];

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

for(let j = 0; j < tabela[i].length; j++){

console.log(tabela[i][j]);

}

}

/\* Saída

1

2

3

4

5

6

7

8

9

10

11

12

\*/

Sem problemas, nosso laço percorreu toda a matriz sem dificuldades...

Vale mencionar que estamos mostrando esse exemplo de matrizes com colunas em quantidade variável, mas não é conveniente produzirmos uma matriz assim. Geralmente, tabelas têm uma semântica nas colunas, ou seja, elas representam algo.

Não podemos omitir colunas se quisermos manter essa semântica, pois os dados se perdem de seu significado se mudarem de coluna. Sendo assim, a tática deve ser usar um valor impossível como um marcador que a coluna está vazia.

O valor impossível depende do problema, lembra do método indexOf que vimos em arrays?

Ele usa o -1 porque no javascript os index são sempre positivos, então -1 não pode de jeito nenhum ser um index válido. Não é o mesmo em outra linguagens como python por exemplo, que permite index negativo.

Via de regra, usar undefined ou null tende a ser a solução mais segura, porque, normalmente, eles não são parte dos valores válido de uma tabela. Por exemplo:

const tabela = [

[1, null, null, 2, 3],

[4, 5, 6, 7, null],

[8, 9, 10, 11, 12]

];

Aqui indicamos que na linha 0 as colunas 1 e 2 não têm valores.

Bem como a última coluna (4) da linha 1.

Na vida real, muitos conjuntos de dados (datasets) não respeitam esse tipo de convenção, usam strings vazias, omitem valores, usam strings como "n/a" ou coisas do gênero para mostrar a ausência de dados.

Por isso fique ligado, cada dataset será seu próprio mundo, e muitas vezes temos que fazer uma rodada de normalização antes de utilizá-los para garantir certos padrões.

Muitas vezes há omissão dos dados e caímos em quantidades incorretas de colunas, portanto cuidado com isso também...

Quer encontrar uns datasets para brincar um pouco? Acesse o [kaggle](https://www.kaggle.com/).

Você já deve imaginar que para fazermos modificações nos dados de uma matriz basta passar os dois endereços, da linha e da coluna, e fazer a atribuição normalmente.

const tabela = [[],[]];

tabela[0][0] = 10;

tabela[0][1] = 20;

tabela[0][2] = 30;

tabela[1][0] = 40;

tabela[1][1] = 50;

tabela[1][2] = 60;

console.log(tabela.join('\n'));

/\*

10,20,30

40,50,60

\*/

Nesse exemplo criamos uma matriz 2x3 vazia, veja que na declaração só temos a informação de que ela tem duas linhas, nada sobre as colunas...

Atribuímos 3 valores por linha para definir que o número de colunas é 3.

Nesse exemplo, usamos o join() para transformar o array em uma string usando o caractere passado por parâmetro como separador dos valores, no caso \n que significa quebra de linha.

Com isso concluímos matrizes. Lembre-se, todas as outras coisas que você aprendeu que se aplicam aos arrays também se aplicam às matrizes, só tenha cuidado porque as linhas são arrays, mas não temos uma estrutura que representa a coluna...

## Últimas palavras sobre arrays e matrizes

Só para concluírmos o assunto de matrizes e arrays, vamos conversar sobre algumas peculiaridades e boas práticas.

Vamos falar de arrays, mas como vimos tudo que se aplica aos arrays se aplica às matrizes.

Primeira coisa, arrays podem misturar diversos tipos de dados. Podemos ter strings, misturadas com números, misturadas com booleanos, misturadas até mesmo com funções.

Sendo assim, a primeira dica é **aprecie a tipagem dinâmica com moderação**, só misture tipos em arrays se realmente for necessário e fizer sentido. Na maioria das vezes, um objeto seria o ideal e não um array.

Segunda dica, **tente não alterar os dados de um array**, mantenha o array original e aplique transformações em uma cópia, ou, melhor ainda, utilize técnicas que criam novos arrays. Logo mais mostraremos um conjunto poderoso dessas técnicas... Só precisamos passar por funções antes...

Terceiro, **arrays de objetos tendem a ser mais interessantes como estratégia do que matrizes**. Raramente uso matrizes nos programas, mas não é difícil que elas sejam a entrada de dados inicial então precisamos saber lidar com elas e transformá-las. Veremos objetos mais a frente também... A vantagem é que objetos preservam sua semântica de dados mesmo quando desvinculados de outra estrutura.

Quarto, já falamos, mas não custa ressaltar, **não modifique o array dentro de um laço que está percorrendo esse mesmo array** e evite hacks comuns como percorrer de trás pra frente, para poder remover dados. Use a estratégia da dica 2.

Quinto, talvez mais importante, **arrays são objetos e como tal são passados por referência**!

Isso quer dizer que fazer arrA = arrB não significa fazer uma cópia de arrB em arrA, isso significa que arrA e arrB apontam para a mesma memória, sendo assim, modificar um é modificar o outro.

Nas linguagens mais antigas chamávamos isso de ponteiros, nas mais modernas de referência.

**Para copiar um array precisamos copiar elemento a elemento.**

## Desestruturação

Só para não dizer que deixamos toda a parte boa para mais para frente vamos mostrar uma sintaxe que pode ajudar um pouco a lidar com as dicas acima...

Na versão ES6, o javascript introduziu um conjunto de sintaxes de desestruturação e restruturação de arrays e objetos, vamos ver apenas a de arrays por enquanto.

### **Spread operator**

O javascript tem um operador chamado spread operator sua sintaxe é ... quando aplicado a um array ele representa os elementos desse array sem a estrutura, ou seja, imagine uma caixinha com 4 bolinhas dentro em fila, agora imagine que a caixa sumiu e ficaram apenas as bolinhas flutuando no ar...

É mais ou menos isso que o spread faz. Ele copia os elementos do array sem copiar o array em si. Você pode pensar, porque raios iríamos querer algo assim??

A vantagem é que podemos montar novos arrays usando o spread e a sintaxe normal de colchetes. Por exemplo, copiar um array é muito fácil usando spread:

const arrA = [1, 2, 3, 4, 5];

const arrB = [ ...arrA];

Simples assim, arrB tem todos os elementos de arrA sem apontar para a sua memória.

Podemos usar essa técnica para adicionar elementos junto com a cópia, por exemplo:

const arrA = [1, 2, 3, 4, 5];

const arrB = [ ...arrA, 6, 7, 8, 9, 10];

Fazendo, montamos um array que além de todos os itens de arrA ainda tem os itens 6, 7, 8, 9 e 10.

Podemos ainda desestruturar o array para um conjunto de variáveis, por exemplo:

const arrA = [1, 2, 3];

const [a, b, c] = arrA;

console.log(a); // 1

console.log(b); // 2

console.log(c); // 3

Se quiser apenas um pedaço do array você pode preservar o restante em uma variável que também será um array:

const arrA = [1, 2, 3, 4, 5];

const [primeiro, ...resto] = arrA;

console.log(primeiro); // 1

console.log(resto); // [2, 3, 4, 5]

Observer o spread antes da variável resto ele que indica que nela estará todo o restante do array que ainda não foi desestruturado...

Geralmente usamos a nomenclatura head e tail quando fazemos isso, sendo head o primeiro valor e tail os demais...

Podemos pegar mais valores antes de usar o spread por exemplo:

const arrA = [1, 2, 3, 4, 5];

const [primeiro, segundo, terceiro ...resto] = arrA;

console.log(primeiro); // 1

console.log(segundo); // 2

console.log(terceiro); // 3

console.log(resto); // [4, 5]

Vale observar que usamos os colchetes após const não para construir um novo array, mas para desestruturar o array original, é diferente do primeiro e segundo exemplos onde de fato estamos declarando um novo array!

Se os colchetes vem logo após const é uma desestruturação. Em outras linguagens essa sintaxe é bastante comum, geralmente aplicada para tuplas, então é capaz que cruze com algo semelhante programando python por exemplo.

Ainda tem mais sobre desestruturação e restruturação de arrays, mas não queremos ir para sintaxes mais confusas, por enquanto paramos por aqui...

Vamos agora para a parte mais importante, as funções, com elas poderemos programar em um estilo chamado programação funcional, de onde vão vir várias das coisas que venho prometendo que veremos mais para frente...

# Funções

Javascript dá uma importância especial para funções que não vemos necessariamente em outras linguagens de programação.

A arquitetura do javascript foi feita de forma que as funções são valores, chamamos isso de "first class citizens" ou cidadãos de primeira classe.

O que quer dizer que funções são valores?

Imagine que todo lugar onde podemos passar um valor, como 10 por exemplo, ou "teste", nós poderíamos passar uma função também.

Sendo assim, podemos colocar funções em variáveis, passá-las como parâmetro para outras funções, retornar novas funções de dentro de uma função e etc...

Isso permite que utilizemos o paradigma/estilo funcional de programação no javascript.

## O que são funções?

Em linguagens que tratam funções com menos importância, elas são apenas trechos de código reaproveitáveis.

Ou seja, escrevemos algumas linhas de código, damos um nome para elas, e quando usamos esse nome em outro código essa linhas executam.

Era muito comum chamar isso de procedimento, rotina, subrotina ou outros nomes nas linguagens mais antigas.

*Exemplo: Hello World*

*//Cria a função*

*function helloWorld(){*

*console.log('hello world');*

*}*

*//Invoca a função*

*helloWord();*

Observe que até o momento, tirando os exemplos dos roladores de dados, usamos diversas funções, mas criamos poucas delas.

Temos dois momentos no uso das funções, o primeiro quando elas são declaradas (criadas) e o segundo quando elas são invocadas. Uma das diversas vantagens de funções é que, uma vez criadas, elas podem ser usadas quantas vezes quisermos.

Como ilustrado acima, criar uma função é fácil, usamos a palavra function seguida do nome da função, seguida de parênteses, e chaves. Dentro das chaves, colocamos os códigos que desejamos rodar quando a função for invocada.

Obs: Nos exemplos abaixo, vamos usar interpolação de strings, no javascript essa sintaxe é chamada de "template string", se não conhecer ainda veja na [documentação no MDN](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/template_strings).

## Parâmetros

Observe o seguinte exemplo:

*Exemplo: Somar*

*function somar() { console.log(`1 + 2 = ${1 + 2}`); }*

*somar() // 1 + 2 = 3*

Essa é uma função de somar, ela produz no console o resultado da soma de 1 com 2.

Sendo assim, sempre que quisermos somar 1 e 2 poderemos invocá-la, super útil, né!?

Nem tanto...

Para tornar funções mais reutilizáveis em diversos casos onde a lógica é a mesma, mas os valores são diversos (como em somas) adicionamos parâmetros às funções, permitindo que elas usem valores escolhidos apenas na hora de invocá-las.

*Exemplo: Somar com parâmetros*

*function somar(num1, num2) {*

*console.log(`${num1} + ${num2} = ${num1 + num2}`);*

*}*

*somar(100, 200) // 100 + 200 = 300*

Criar parâmetros é fácil, basta colocar nomes (como variáveis) entre parênteses na hora de criar sua função. Você pode colocar quantos precisar desde que separados por vírgula.

Quando sua função for chamada (invocada) basta passar os valores dos parâmetros entre parênteses que eles serão enviados para as "variáveis" que criamos na função.

No exemplo acima, somar() é invocado passando os valores 100 e 200 como parâmetros, como a ordem dos parâmetros é respeitada, num1 terá o valor 100 e num2 terá o valor 200.

No javascript, nada impede de que uma função seja invocada com parâmetros faltando, ou de tipos incorretos, então tome cuidado quando estiver criando funções para outras pessoas utilizarem, pois os parâmetros podem vir inconsistentes. É necessário fazer validação e gerar erros caso os parâmetros estejam inconsistentes.

Funções em javascript são naturalmente polimórficas, isso quer dizer que elas permitem que em seus parâmetros aceitem qualquer tipo de dados. Isso quer dizer que nossa função de soma acima funcionaria com float, com int, e até com strings, porque todos esses tipos têm o operador +.

Veja:

somar(10.5,20.7); // 10.5 + 20.7 = 31.2

somar(10, 20); // 10 + 20 = 30

somar("isso é ", "um teste!"); // isso é + um teste! = isso é um teste!

No entanto, isso também quer dizer que ela pode ser chamada como tipos que não possuem o operador + e isso produzirá um erro ou se comportará de forma imprevisível.

## Retorno

Na maioria dos casos, desejamos obter da função algum valor resultante e utilizá-lo em nosso código. A função de soma acima é útil para fazer apenas uma soma, mas se quisermos utilizá-la no meio de uma conta mais complexa isso não é possível, afinal, tudo o que ela faz é mostrar o resultado na tela.

Podemos modificá-la para que, invés de mostrar o resultado na tela, ela devolva ao código que a invocou o resultado. Isso também é muito fácil de fazer, basta no final da função usar o comando return e indicar logo após o que você deseja retornar.

*Exemplo: Somar com parâmetros e retorno*

*function somar(num1, num2) {*

*return num1 + num2;*

*}*

*const result = somar(100, 200);*

*console.log(result); //300*

Sua função pode fazer o que você quiser, mas só pode retornar um valor!

No entanto, nada te impede de retornar um array, um objeto ou uma nova função...

## Return como finalizador da função

Sempre que sua função encontra um return ela termina imediatamente. Sendo assim, nada após o return irá executar. Por isso que sempre colocamo-os na última linha.

No entanto, você pode usar isso em seu favor para interromper prematuramente uma função.

Por exemplo, na nossa soma podemos validar os parâmetros e se não for o que esperamos podemos retornar imediatamente NaN que representa "not a number", para isso usaremos a função isNaN().

*Exemplo: Somar com parâmetros e retorno se os parâmetros são números*

*function somar(num1, num2) {*

*if(isNaN(num1) || isNaN(num2)) return NaN;*

*return num1 + num2;*

*}*

Ou com ternário:

*Exemplo: Somar com parâmetros e retorno se os parâmetros são números (ternário)*

*function somar(num1, num2) {*

*return (isNaN(num1) || isNaN(num2)) ? NaN : num1 + num2;*

*}*

## Funções Anônimas

Se quisermos fazer uma função que vai rodar apenas uma vez ou que vai ser passada como parâmetro para outra função podemos criá-la sem nome...

Imagine o seguinte, tenho uma página que tem um botão, assim que a página carregar desejo adicionar um evento ao botão que mostre no console que ele foi clicado.

*Não se preocupe com as funções usadas nesse exemplo, chegaremos nelas eventualmente e explicaremos direitinho... por enquanto foque apenas na função anônima...*

O javascript têm uma função chamada addEventListener que faz isso, ele espera que passemos por parâmetro a função que desejamos invocar quando o botão for clicado, para isso podemos usar uma função anônima (nossa função deve ter um parâmetro, evt, que representa o evento de click):

const button = document.getElementById('meubotao');

button.addEventListener(function(evt){ console.log('click'); });

Podemos colocar todo esse código em uma função anônima que se invoca imediatamente após a declaração também, seria assim:

(function(){

const button = document.getElementById('meubotao');

button.addEventListener(function(evt){ console.log('click'); });

})();

Aqui, colocamos a função toda entre parênteses e um par adicional de parênteses para invocá-la. Isso faz com que ela rode imediatamente e deixe de existir.

Essa sintaxe não é da mais comuns, mas pode ser útil...

Uma forma muita mais comum de usar funções anônimas é colocá-las em variáveis, lembre-se que funções são valores e podem ser armazenadas em variáveis:

const somar = function(num1, num2) { return num1 + num2; }

Veja, criamos uma função anônima e colocamos dentro das constante somar. Isso produz o mesmo efeito de dar o nome somar para a função. Para usá-la não muda nada somar(1,2) por exemplo.

Sempre use const para garantir que sua função não seja sobrescrita por outra em uma atribuição incorreta...

Essa forma de escrever funções costuma ser bem recorrente no javascript, eu mesmo prefiro essa forma do que a padrão, mas gosto ainda mais de arrow functions...

## Arrow Functions

Funções anônimas, sejam elas armazenadas em variáveis ou não, são tão comuns em javascript que temos uma forma simplificada de criá-las, se trata de uma nova sintaxe introduzida no ES6 chamada "arrow function" em outras linguagens eles podem ter no nome "lambda" por conta do cálculo lâmbda que deu origem à programação funcional.

Arrow functions são escritas sem a palavra function, e a quando tem apenas um comando que será retornado as chaves podem ser omitidas, bem como a palavra return. Se apenas um parâmetro for usado os parênteses podem ser omitidos também. Separando os parâmetros do corpo da função termos uma flecha => por isso o nome arrow function.

Exemplo: Somar com parâmetros e retorno como arrow function

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

Observe que, por conta de todas as omissões que essa sintaxe nos permite fazer, funções simples são escritas de forma muito prática e reduzida.

No exemplo, o return está implícito, as chaves não são necessárias porque só há um comando. Quando temos mais de um parâmetro ou nenhum, precisamos manter os parênteses, então isso não pudemos tirar...

Vamos rescrever o exemplo do botão com arrow function para exemplificar mais:

const button = document.getElementById('meubotao');

button.addEventListener(evt => console.log('click'));

Nesse caso, omitimos os parênteses, e as chaves, a função ficou super reduzida.

Arrow functions maiores podem ser escritas mantendo as chaves, mas lembre-se, sempre que usar chaves você não poderá mais omitir a palavra return para retornar...

const teste = () => {

console.log('1');

console.log('2');

console.log('3');

console.log('testando...');

return "OK";

}

Nesse exemplo a função teste não tem parâmetros, simbolizamos isso usando parênteses vazios antes da flecha. Somos obrigados a manter o return porque temos chaves e nela várias expressões, sendo assim, precisamos indicar qual será a que retorna...

E com isso terminamos o básico sobre funções, o próximo conteúdo é sobre o this dentro de funções, um problema caótico do javascript, pode considerar um conteúdo opcional, mas um dia você vai ter que recorrer a ele. Em seguida veremos objetos, e encerramos o javascript básico...

***!****O tópico a seguir provavelmente só será útil para você quando tiver algum dos problemas que descrevemos nele, sendo assim, dê uma lida por cima por enquanto e saíba que pode recorrer a esse conteúdo caso tenha esses problemas, aqui explicamos o porque e como resolver problemas usando this dentro de funções...****!***

## A maldição do this

Dizem que toda linguagem tem uma decisão de arquitetura que parecia uma boa ideia como um feature da linguagem, mas acaba se mostrando um grande erro que gera mais caos do que qualquer outra coisa.

No caso do JAVA e do C# e outras consideramos o null como esse erro.

No caso do javascript considero o this nas funções.

Claro que um monte de gente vai defender esse this e mostrar como ele é útil e etc... e de fato existe seu uso em objetos como fazemos em orientação a objetos em diversas linguagens e ninguém está reclamando desse uso. Se fosse apenas ele não haveria problemas. Nos demais casos, aplicação parcial resolveria sem tanto caos...

Se você perguntar o que é o this em uma função não há uma resposta melhor do que "depende".

Isso se dá porque ele pode ser absolutamente qualquer coisa. O navegador passa um objeto como this, mas uma biblioteca pode passar algo diferente no lugar, e podemos nós mesmos alterar o this de uma função antes ou ao invocá-la.

Sendo assim, nunca temos certeza absoluta do que terá no this de uma função. Essa incerteza gera muitos problemas e soluções estranhas como const self = this; quando você tem que lidar com vários this diferentes em momentos distintos na mesma função.

Até ter problemas com this vai ser difícil você entender o escopo do caos que ele provoca, fizemos esse tópico para te alertar que isso vai acontecer mais cedo ou mais tarde e para mostrar três funções que te permitem manipular o this caso ele não seja o que você esperava...

### **Bind, Call e Apply**

Para piorar as coisas this muda de valor de acordo com o contexto.

Por exemplo, no contexto global, this se refere ao objeto Window do navegador.

*Exemplo: this em contexto global (não se aplica ao node.js)*

*Console.log(this); // Window*

Dentro de um objeto, this refere-se ao próprio objeto:

*Exemplo: this em um objeto*

*const obj = {*

*num : 10,*

*printNum : function(){ console.log(this.num); }*

*}*

*obj.printNum(); // 10*

Aqui usamos o this.num como sendo "desse mesmo objeto" pegue o valor de num.

No entanto, se copiarmos essa função para o contexto global this voltará a ser Window.

*Exemplo: Mudança de contexto alterando this*

*const printNum = obj.printNum;*

*printNum(); // undefined*

### **bind()**

Como tiramos a função do contexto do objeto, o this mudou e não conseguimos mais ver o valor interno do objeto. Podemos remediar isso usando bind.

*Exemplo: alterando this manualmente pelo bind()*

*const printNum = obj.printNum.bind(obj);*

*printNum(); // 10*

Quando chamamos o bind() de uma função, ele criará um nova função, o que for passado no primeiro parâmetro será o novo valor do this para essa nova função. Aqui, mudamos na mão de Window (padrão do contexto global), para o próprio objeto obj. Se passarmos outros parâmetros para o bind() ele usará esses valores como parâmetros da função original.

*Exemplo: aplicação parcial com bind() (ignorando o this)*

*const somar = function(n1, n2){ return n1 + n2; }*

*const somar1 = somar.bind(null, 1);*

*const result = somar1(2);*

*console.log(result); // 3*

Nesse caso, não nos importamos com o this então passamos null, usamos o bind para fazer com que o primeiro parâmetro da nova função criada a partir de somar() seja sempre 1. Agora só precisamos passar o segundo parâmetro.

Usamos o bind() para especializar a função de soma para sempre somar 1 em outro número.

Logo, mais mostraremos "aplicação parcial", que faz a mesma coisa que esse bind de um jeito mais prático, então não se preocupe muito com esse exemplo, ele é só para ilustrar que o bind é capaz de fazer isso...

## call()

Como vimos, bind() cria uma versão modificada de uma função com o this sobrescrito e com valores padrão para os parâmetros, se desejar.

No entanto, as vezes só queremos chamar a função corrigindo o this e não criar uma nova.

Para isso, basta usar o método call(). Ele é idêntico ao bind() em assinatura, o primeiro parâmetro é o novo valor do this e os demais são os parâmetros da função.

A única diferença é que ele não cria uma função nova.

*Exemplo: Usando call() para alterar o this e passar parâmetros*

*const somar = function(n1, n2){ return n1 + n2; }*

*const result = somar.call(null, 1, 2);*

*console.log(result); // 3*

## apply()

apply() tem apenas uma diferença com relação ao call() que é que os parâmetros da função são passados como um array e não um a um separados por vírgula.

*Exemplo: Usando apply() para alterar o this e passar parâmetros como array*

*const somar = function(n1, n2){ return n1 + n2; }*

*const valores = [10, 20];*

*const result = somar.apply(null, valores);*

*console.log(result); // 30*

Hoje, isso é totalmente desnecessário, pois, como vimos temos o spread operator que nos permite fazer isso com facilidade, se o this não importa.

*Exemplo: Alternativa ao apply() sem alterar o this*

*const somar = function(n1, n2){ return n1 + n2; }*

*const valores = [10, 20];*

*const result = somar(...valores);*

*console.log(result); // 30*

Se o this importar podemos usar o spread com o call(), sendo assim apply() nem precisa mais ser usado...

*Exemplo: Alternativa ao apply() alterando o this*

*const somar = function(n1, n2){ return n1 + n2; }*

*const valores = [10, 20];*

*const result = somar.call(null, ...valores);*

*console.log(result); // 30*

### **Arrow Functions e this**

Você já sabe que a sintaxe que prefiro para funções é a de arrow function, mas deve ter percebido que desde que começamos a falar de this não usei nenhuma arrow function...

Isso se dá porque arrow functions são imunes às alterações do this, sendo assim sempre que desejamos preservar o this do jeito que está podemos usá-las.

No entanto, isso é uma limitação delas também, contextos em que o this modificado é necessário não admitem o uso de arrow functions... Veremos isso em orientação a objetos no javascript no futuro.

# Objetos

Chegamos ao último tópico da lógica de programação básica em javascript.

Nesse tópico, veremos o básico sobre objetos e sua sintaxe, depois veremos em mais profundidade em orientação a objetos.

Objetos são a melhor forma de expressarmos dados complexos em programação.

Com o que vimos até o momento, para fazer uma agenda, por exemplo, precisaríamos ter dois arrays, um para o nome e outro para o telefone ou uma matriz.

No caso dos arrays, o "contato" da agenda seria o index, ou seja, a posição nos arrays em que o nome e o telefone são da mesma pessoa.

Observe:

*Exemplo: Agenda (constante) criada com arrays*

*const nomes = ["contato1", "contato2", "contato3"];*

*const telefones = ["000000000", "111111111", "222222222"];*

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

*console.log(nomes[i], telefones[i]);*

*}*

*/\**

*Saída:*

*contato1 000000000*

*contato2 111111111*

*contato3 222222222*

*\*/*

Nesse exemplo representamos a agenda usando um array para cada informação.

Vários problemas podem acontecer nesse exemplo, uma vez que o contato é o index desses arrays.

Quando dizemos nomes[0] e telefones[0] estamos pegando os dados do primeiro contato.

É muito fácil cometer um deslize que "desalinhe" os arrays, por exemplo, inserir um contato novo sem telefone.

Esse desalinhamento é o pior dos casos aqui, pois todos os dados de todos os contatos que vierem depois serão corrompidos...

Para manter a integridade dos dados, precisamos adicionar null ou algum valor que represente vazio se o contato não tiver telefone.

Ainda assim, erros mais simples podem ocorrer, como incrementar incorretamente o index e imprimir o nome do contato e o telefone de outro contato.

Se alterarmos o programa para usar uma matriz, as coisas melhoram um pouco:

*Exemplo: Agenda (constante) criada com matriz*

*const agenda = [*

*["contato1", "000000000"],*

*["contato2", "111111111"],*

*["contato3", "22222222"]*

*];*

*for(contato of agenda){*

*console.log(contato[0], contato[1]);*

*}*

*/\**

*Saída:*

*contato1 000000000*

*contato2 111111111*

*contato3 222222222*

*\*/*

Observe que nesse caso, nome e telefone estão juntos no mesmo index de agenda.

Dizer agenda[0] nos dá tanto o nome quanto o telefone do primeiro contato, sendo assim, temos mais garantia de integridade.

Também pudemos utilizar o for-of e dar alguma semântica ao programa chamando cada valor de "contato".

No entanto, ainda resta uma problema, obtemos os dados do contato pelos index do array interno, sendo assim, temos que fazer contato[0] e contato[1].

Sendo assim, temos que saber as posições dos dados e o que elas representam...

Além disso, ambos exemplos escalam mal, se quisermos adicionar mais dados sobre um contato, teremos algumas dificuldades com a matriz e enormes dificuldades com os arrays.

Nas matrizes, precisamos pensar em colocar valores nulos para todos que não tem o novo dado, pois precisamos manter coerente os index do contato, bem como uma quantidade igual de "colunas" em cada linha.

No caso dos arrays é pior, teremos que criar novos arrays e manter a consistência dos index em todos. Um erro em um, e todos os contatos terão aquele dado corrompido.

Isso sem nem mencionar que para colocar tudo isso em um banco de dados, ou carregar os dados a partir de um, seria bem problemático...

## Objetos

Nunca perca de vista o que dissemos lá no início, linguagens são baseadas em sistemas de tipos.

Isso quer dizer que toda as linguagens têm tipos já estabelecidos, mas também quer dizer (na maioria esmagadora das linguagens) que podemos compor novos tipos.

Criar um tipo novo em uma linguagem é fazer uma composição dos tipos que a linguagem já tem. Por exemplo, nosso contato, como está, é a composição de duas strings.

Cada linguagem tem seu jeito de criar tipos algumas chamam de typedef, structs, classes, e por aí vai...

No javascript, por ser uma linguagem dinâmica, temos algumas formas de construir tipos.

Podemos simplesmente escrever objetos diretamente conforme a necessidade ou podemos criar classes ou ainda funções construtoras.

Nesse tópico abordaremos apenas a primeira forma e funções construtoras para simplificar um pouco o código.

Em orientação a objetos, abordaremos classes.

Vamos começar rescrevendo o exemplo acima usando objetos e depois discutiremos a sintaxe:

*Exemplo: Agenda (constante) criada com objetos*

*const agenda = [*

*{ nome : "contato1", telefone : "000000000"},*

*{ nome : "contato2", telefone : "111111111"},*

*{ nome : "contato3", telefone : "222222222"}*

*];*

*for(contato of agenda){*

*console.log(contato.nome, contato.telefone);*

*}*

*/\**

*Saída:*

*contato1 000000000*

*contato2 111111111*

*contato3 222222222*

*\*/*

Observe que partimos do exemplo da matriz, mas substituímos os arrays internos por um objeto para cada contato.

Nos objetos, usamos chaves para indicar que escreveremos um objeto e quantos pares chave-valor desejarmos separados por vírgula.

Os pares chave-valor, por sua vez, são separados por dois pontos :.

Cuidado para não usar = no lugar de :.

Se você conhece alguma linguagem que utiliza dicionários os conceitos são muito semelhantes...

Mas vamos lá, que vantagens temos nessa abordagem?

Com relação ao exemplo dos arrays, que era o pior, temos muitas vantagens, a primeira é ter apenas um array, a segunda que todos os dados estão contidos no objeto, então não há mais possibilidade de ter problemas com os index ou de que um erro destrua a integridade de vários dados agora, no máximo, podemos estragar os dados de um dos contatos.

Com relação ao exemplo com a matriz, nos livramos daquele index para escolher a informação que desejamos, a ordem das informações não faz mais diferença pois agora indexamos-as por um nome (chave) e não mais por uma posição.

O programa agora é mais fácil de ler e modificar, pois usamos o nome da informação. É muito mais fácil ler e entender contato.nome do que contato[0].

Novas informações podem ser adicionadas sem dificuldades e podem ser nulas ou nem mesmo aparecer nos objetos sem comprometer as demais, pois não há ordem.

Por padrão, nos objetos uma chave inexistente retornará undefined, portanto, podemos adicionar um e-mail por exemplo apenas em um contato, e isso não fará o programa dê erro quando um contato não tiver o e-mail, apenas mostrará undefined. Não era esse o caso com arrays e matrizes, onde tentar ler um valor de uma posição que não existe daria erro...

*Exemplo: Agenda (constante) criada com objetos, com email opcional*

*const agenda = [*

*{ nome : "contato1", telefone : "000000000", email : "contato1@teste.com"},*

*{ nome : "contato2", telefone : "111111111"},*

*{ nome : "contato3", telefone : "222222222"}*

*];*

*for(contato of agenda){*

*console.log(contato.nome, contato.telefone, contato.email);*

*}*

*/\**

*Saída:*

*contato1 000000000 contato1@teste.com*

*contato2 111111111 undefined*

*contato3 222222222 undefined*

*\*/*

Se preferir você pode tratar os emails não preenchidos para que não mostre undefined na tela.

*Exemplo: Agenda (constante) criada com objetos, com email opcional tratando a saída*

*const agenda = [*

*{ nome : "contato1", telefone : "000000000", email : "contato1@teste.com"},*

*{ nome : "contato2", telefone : "111111111"},*

*{ nome : "contato3", telefone : "222222222"}*

*];*

*for(contato of agenda){*

*console.log(contato.nome, contato.telefone, contato.email || "");*

*}*

*/\**

*Saída:*

*contato1 000000000 contato1@teste.com*

*contato2 111111111*

*contato3 222222222*

*\*/*

Usamos o "ou" || nesse caso, quando os valores não são booleanos o "ou" avalia se são truthy ou falsy, se forem truthy ele usa o valor, se forem falsy ele usa o valor que estiver do seu lado direito. Você pode pensar que é um operador para valores padrão, se há valor use, senão use esse valor padrão.

Sendo assim, contato.email || "" é o mesmo que o ternário contato.email ? contato.email : "" em uma sintaxe reduzida.

Se você preferir, em vez de escrever os objetos você pode fazer uma função construtora, que basicamente é uma função que produz um objeto.

*Exemplo: função construtora para contato*

*const createContato = (nome, telefone, email) => ({*

*nome : nome,*

*telefone : telefone,*

*email : email || ""*

*});*

A função createContato já recebe o nome, telefone e email por parâmetro e retorna um objeto pronto, até o tratamento do e-mail opcional já foi feito.

Usei arrow function então o return está implícito, os parenteses depois da flecha são para indicar que a abertura de chaves é um objeto e não o corpo da função, não esqueça deles!

Observe o padrão repetitivo: nome : nome, telefone : telefone e etc...

Quando o nome da chave e a variável ou no caso o parâmetro que contém seu valor tem o mesmo nome podemos simplificar, por exemplo:

*Exemplo: chaves com mesmo nome que variáveis*

*const nome = "teste";*

*const aluno = {nome : nome};*

*Pode se tornar apenas:*

*const nome = "teste";*

*const aluno = {nome};*

Veja que em vez de nome: nome usamos apenas nome ele usará o nome da variável como chave.

Sendo assim, nossa função pode ser simplificada para:

const createContato = (nome, telefone, email) => ({nome, telefone, email : email || "" });

Para utilizá-la basta passar os valores:

const contato1 = createContato("contato1", "000000000", "contato1@teste.com");

const contato2 = createContato("contato2", "111111111", "");

const contato3 = createContato("contato3", "222222222");

Observe que o e-mail foi passado no primeiro contato, no segundo passamos string vazia e último omitimos completamente o parâmetro e-mail. Todos funcionam.

Para esse exemplo dos contatos não temos grandes vantagens em usar a função construtora porque a única lógica especial na criação desse objeto é tratar o email como opcional, no entanto, em objetos mais complicados de construir ela pode ser uma grande aliada!

Inclusive é um design pattern chamado Factory Method...

Objetos são muito mais poderosos do que vimos até o momento, vamos abordá-los em detalhes e ver suas outras capacidades quando trabalharmos com orientação a objetos.

Com isso encerremos a lógica de programação em javascript.

Seguimos agora para programação funcional, que é um paradigma/estilo de programação baseado em funções, onde veremos novas formas de usar funções e aquelas funções de arrays que prometemos nos tópicos anteriores

# Programação Funcional

Programação funcional é um paradigma/estilo de programação baseado em conceitos matemáticos que é tão antigo quanto os demais estilos, no entanto, tem ganhado grande popularidade nos últimos tempos.

Não vamos abordar programação funcional a fundo, mas seus preceitos básicos e técnicas mais utilizadas em javascript.

## Benefícios

Não quero programar no estilo funcional, o que ganho de saber essas coisas?

Mesmo que seu paradigma de escolha seja estruturado ou orientado a objetos, a programação funcional tem conceitos e cuidados que podem ser aplicados nos demais estilos para produzir códigos melhores, mais simples e escaláveis.

Os conceitos funcionais já invadiram linguagens orientadas a objetos para dar alguns exemplos:

No C# temos o linq que é uma das principais ferramentas para tratamentos de dados que o framework .net oferece e ele é todo baseado em conceitos funcionais. Bem como, Tasks que lidam com contexto assíncrono também são baseadas em monadas funcionais.

O JAVA foi resistente ao conceitos funcionais, por mais tempo, mas no java 8 implementou lambdas que são pré-requisitos para programar funcional.

No javascript, hoje a forma padrão de manipulação de dados em arrays são as formas funcionais. A programação com constantes e imutabilidade também são conceitos funcionais, a programação assíncrona com Promises também é baseada na monada Task da programação funcional.

## Pré-requisitos

Para uma linguagem ser adequada para o uso de programação funcional ela precisa no mínimo de funções tratadas como cidadãos de primeira classe.

Já falamos, mais isso quer dizer que funções são tratadas como valores, podendo ser armazenadas em variáveis e passadas por parâmetro ou retornadas por funções.

Já vimos que o javascript atende esse requisito.

Outros conceitos são desejáveis para facilitar nossa vida funcional. Por exemplo, pattern matching, currying, tipos algébricos, composição, pipe e etc...

No javascript não temos nada disso nativamente então criaremos ferramentas para lidar com alguns desses casos conforme necessário. Também podemos utilizar bibliotecas para programação funcional. Mas nem tudo dá para fazer.

Se quiser uma experiência funcional completa você precisará de uma linguagem específica para programação funcional como F# (irmão funcional do C#), Scala (baseado em JAVA), Clojure (usada no NuBank), Erlang (para concorrência e programação paralela) ou Haskell (o mais acadêmico e com maior curva de aprendizado).

No entanto, no javascript poderemos trabalhar todos os conceitos básicos da funcional e tornar nossos códigos melhores para as necessidades do dia a dia.

## Conceitos básicos

Trabalharemos os seguintes conceitos:

* Funções de Alta Ordem
* Pointfree
* Aplicação Parcial
* Imutabilidade
* Funtores (básico)
* map()
* filter()
* reduce()
* Composição

## Preceitos básicos

Na programação funcional teremos duas estruturas objetos e morfismos. Em outras palavras, ou uma coisa é uma estrutura de dados ou uma transformação de uma estrutura de dados, não há outras opções.

Objetos podem ser os objetos que acabamos de aprender ou constantes de qualquer tipo.

Morfismos serão funções.

Sendo assim, objetos armazenarão nossos valores e funções modificarão esses valores.

Modificações só podem ser feitas por funções, então não poderemos mais utilizar o operador = para modificar valores. Se os valores não podem ser modificados, não há porque existir variáveis, então usaremos apenas constantes.

Sem variáveis, os laços de repetição também não fazem sentido (todos dependem de variáveis), então usaremos outras técnicas.

Sempre optaremos por criar complexidade por meio da composição de coisas simples e toda a complexidade estará em funções pois são elas que admitem composição.

Funções que admitem composição só podem ter um parâmetro, então usaremos aplicação parcial para transformar funções de vários parâmetros em funções com apenas um parâmetro.

Não vou me estender mais, mas você percebe que os conceitos acima estão aparecendo aqui, um a um?

Funções de alta ordem, pointfree e aplicação parcial fundamentam ou apoiam a composição.

Map, filter e reduce serão nossos substitutos para laços e tudo vai se encaixando.

## Pureza e Efeitos Colaterais

O conceito de pureza é simples:

Uma função é pura quando sempre produz o mesmo retorno quando passados os mesmos parâmetros.

Por exemplo, a função somar() sempre retornará 3 quando os parâmetros forem 1 e 2.

A função somar() não depende de nenhuma informação de fora dela além dos números que ela precisa somar, por isso, não há nenhuma forma de seu retorno mudar porque o contexto em que ela está rodando mudou. Ela é auto-suficiente.

Sendo assim, podemos copiar essa função para qualquer programa que estivermos fazendo sem medo, ela funcionará como esperado sempre! Essa é uma grande vantagem das funções puras, sua reutilização é garantida.

Outra propriedade interessante é que se compusermos funções puras que funcionam em uma nova função, essa função funcionará corretamente. Ela nem precisa ser testada, se a composição foi feita corretamente (na ordem certa), ela funcionará como esperado.

Infelizmente, nem tudo pode ser escrito por meio de uma função pura.

Qualquer coisa que dependa do ambiente tornará uma função impura, por exemplo, sortear um número aleatório, escrever no console, ler um dado do disco ou banco de dados, e por aí vai...

Todas essas operações nós chamamos de efeitos colaterais. São coisas que podem alterar a forma como a função produz seu resultado impedindo que elas sejam consideradas puras.

Quando funções não retornam ou não têm parâmetros temos uma grande indicação de impureza, pois elas devem depender de dados externos ou produzir efeitos colaterais, porque se não for isso a função não faria nada...

Existem funções puras sem parâmetros, funções constantes, que sempre retornam o mesmo resultado, mas não há funções puras sem retorno.

Lembre-se, na funcional, funções são transformações de valores, se elas não retornam o resultado da transformação e não produziram efeitos o que elas fizeram?

Sendo assim, funções puras sempre retornam e, se não forem constantes, sempre possuem parâmetros.

Para programar para o mundo real, usaremos funções puras e impuras, só tente entender em que categoria sua função se encaixa para saber o que você pode ou não fazer com ela.

Por exemplo, na imensa maioria das vezes, funções impuras não funcionarão corretamente quando copiadas e coladas de um projeto para o outro e sempre são mais complicadas para serem testadas por técnicas como testes unitários, pois o ambiente onde elas estão influencia sua execução.

Vamos começar pelo pré-requisito de todas as técnicas que são as funções de alta ordem.

# Funções de alta ordem (High Order Functions)

O conceito aqui é muito simples, funções de alta ordem são funções que recebem outras funções como parâmetro ou que retornam novas funções.

Vamos começar criando algumas funções matemáticas simples:

const somar = (a, b) => a + b;

const subtrair = (a, b) => a - b;

const multiplicar = (a, b) => a \* b;

const dividir = (a, b) => b > 0 ? a / b : NaN;

Aqui temos as 4 operações básicas, todos os returns foram omitidos porque usamos arrow functions. A única um pouco mais complexa é a divisão, onde usamos um ternário para evitar divisões por zero, retornando "not a number" nesse caso.

Agora criaremos uma função de alta ordem que recebe uma operação (uma das funções acima) e os dois números e faz a conta.

const calcular = (operacao, a, b) => calcular(a,b);

O parâmetro operacao de calcular é uma função, ela é invocada usando a e b como seus parâmetros.

Para utilizar a função calcular é simples, basta passar por parâmetro a função desejada e dois números.

const resultado = calcular(somar, 1, 2);

console.log(resultado); // 3

Mas qual é a vantagem?

Isso simplificaria um programa em que a escolha da operação é feita pelo usuário por exemplo.

Imagine que podemos ter 3 variáveis, valor1, valor2 e operação. Quando o usuário clica no botão de um uma operação (+ - \* /) já colocamos a função correta na variável operação e depois aplicamos com calcular() a operação escolhida, isso nos polparia um monte de condicionais.

O mais importante aqui é observar que somar foi passado como um função, mas não invocado (não há parênteses).

Se procurarmos fazer um software priorizando esse tipo de sintaxe, tentando ao máximo delegar a invocação das funções e a passagem de parâmetros para outras estruturas teremos um estilo de programação chamado pointfree.

Podemos também criar outra funções em termos de calcular() por exemplo:

const media = (a,b) => calcular(dividir, calcular(somar, a, b), 2.0);

//ou

const desconto = (preco) => calcular(subtracao, preco, calcular(multiplicar, preco, 0.1));

Não vou dizer que são as formas mais fáceis de criar essas funções, mas é possível expressar qualquer operação complexa usando calcular() sem usar os parênteses de precedência e sem usar nenhum operador, apenas funções.

Em alguns momentos, trocar operadores binários como + por uma função pode ser mais apropriado, especialmente quando trabalhamos com árvores. Se quiser, dê uma olhada em operadores infix, prefix e postfix, mas isso é tópico de estrutura de dados que não trataremos aqui...

## Retornando funções

Podemos fazer funções de alta ordem para retornar outras funções, por exemplo, imagine que você tem a mesma função em dois softwares diferentes, ela teria que fazer a mesma coisa, no entanto seu retorno, em um dos softwares será um JSON e no outro um XML.

Podemos fazer uma função de alta ordem produzir duas versões diferentes de uma função para adequarem o retorno ao programa que as invoca, assim não precisamos criar duas funções diferentes, uma em cada programa, que façam a mesma coisa.

*const createFormatContatoFor = (software) => {*

*if(software === "software1") return contato => JSON.stringify(contato);*

*if(software === "software2") return contato => `*

*<contato>*

*<nome>${contato.nome}</nome>*

*<telefone>${contato.telefone}</telefone>*

*</contato>*

*`*

*}*

*No software 1:*

*const formatContato = createFormatContatoFor("software1");*

*const contato = { nome: "teste", telefone: "000000000" };*

*console.log(formatContato(contato));*

*/\*Saída*

*{"nome":"teste","telefone":"000000000"}*

*\*/*

*No software 2:*

*const formatContato = createFormatContatoFor("software2");*

*const contato = { nome: "teste", telefone: "000000000" };*

*console.log(formatContato(contato));*

*/\*Saída*

*<contato>*

*<nome>teste</nome>*

*<telefone>000000000</telefone>*

*</contato>*

*\*/*

*Vale mencionar que o XML acima não está aderente às regras de formatação é apenas ilustrativo.*

No exemplo, createFormatContatoFor() utiliza o parâmetro software para decidir qual das duas versões utilizar, a que formata em JSON ou a que formata em XML.

A função formatContato() é o resultado dessa decisão. Nela não há nenhum vestígio que ela foi produzida pela lógica interna de outra função, em todos os sentidos é uma função normal como qualquer outra.

Vale mencionar que as funções podem ser declaradas dentro de outras funções no javascript, não apenas retornadas.

Também é interessante apontar que as funções internas têm acesso aos parâmetros, variáveis e constantes da função maior. Isso se dá por um conceito chamado "closure".

No nosso exemplo, se as funções internas precisassem do valor do parâmetro software para alguma coisa, poderiam utilizá-lo sem problemas.

Logo mais veremos o que é aplicação parcial, com ela podemos utilizar o conceito que acabamos de ver para produzir funções especializadas a partir de funções mais genéricas.

Mas antes disso, vamos cumprir a promessa de mostrar os métodos de array, que são funções de alta ordem, que podemos utilizar para facilitar nossa vida ao trabalhar com essa estrutura de dados.

Crie um vetor qualquer e faça um programa que imprima cada elemento do vetor usando o for.

## 2

Faça um programa que imprima todos os itens de um vetor usando while e compare com o exercício 1.

## 3

Faça um programa que peça para o usuário digitar um número n e imprima um vetor com todos os números de 0 a n-1.

Exemplo: se o usuário digitar 5, o programa deve imprimir [0, 1, 2, 3, 4]

## 4

Faça um programa que olhe todos os itens de um vetor e diga quantos deles são pares.

## 5

Faça um programa que imprima o maior número de um vetor.

## 6

Faça um programa que, dadas dois vetores de mesmo tamanho, crie um novo vetor com cada elemento igual a soma dos elementos do vetor 1 com os do vetor 2, na mesma posição.

Exemplo:

Dadas vetor1 = [1, 4, 5] e vetor2 = [2, 2, 3], então vetor3 = [1+2, 4+2, 5+3] = [3, 6, 8]

## 7

Faça um programa que dados dois vetores de mesmo tamanho, imprima o produto escalar entre eles.

OBS: produto escalar é a soma do resultado da multiplicação entre o número na posição i do vetor1 pelo número na posição i do vetor2.

## 8

Faça um programa que pede para o usuário digitar 5 números e, ao final, imprime um vetor com os 5 números digitados pelo usuário (sem converter os números para o tipo number).

Exemplo: Se o usuário digitar 1, 5, 2, 3, 6, o programa deve imprimir o vetor [‘1’,‘5’,‘2’,‘3’,‘6’]

## 9

Pegue o vetor gerada no exercício anterior e transforme cada um dos itens desse vetor em um number.

OBS: Não é para alterar o programa anterior, mas sim o vetor gerado por ele.

## 10

Faça um programa que peça as 4 notas bimestrais e mostre a média aritmética delas, usando vetores.

## 11

Sorteie um vetor de 10 números e imprima:

a. um vetor com os 4 primeiros números;

b. um vetor com os 5 últimos números;

c. um vetor contendo apenas os elementos das posições pares;

d. um vetor contendo apenas os elementos das posições ímpares;

e. um vetor inverso do vetor sorteado (isto é, um vetor que começa com o último elemento o vetor sorteado e termina com o primeiro);

f. um vetor inverso dos 5 primeiros números;

g. um vetor inverso dos 5 últimos números.

## 12

Faça um programa que sorteia 10 números entre 0 e 100 e conte quantos números sorteados são maiores que 50.

## 13

Faça um programa que sorteie 10 números entre 0 e 100 e imprima:

a. o maior número sorteado;

b. o menor número sorteado;

c. a média dos números sorteados;

d. a soma dos números sorteados.

## 14

Desafio 1 - Faça um programa que peça para o usuário digitar o nome e a idade de um aluno e o número de provas que esse aluno fez. Depois, o programa deve pedir para o usuário digitar as notas de cada prova do aluno. Ao final o programa deve imprimir um vetor contendo:

a. Nome do aluno na posição 0;

b. Idade do aluno na posição 1;

c. Um vetor com todas as notas na posição 2;

d. A média do aluno na posição 3;

e. true ou talse, caso a média seja maior que 5 ou não, na posição 4.

Dica: Use o que você fez nos exercícios anteriores para criar esse programa.

## 15

Desafio 2 - Faça um programa como o do item anterior, porém que imprima a média sem considerar a maior e menor nota do aluno (nesse caso o número de provas precisa ser obrigatoriamente maior que dois).

Dica: crie uma cópia com o vetor de todas as notas antes de fazer a média.

## 16

Desafio 3 - Faça um programa que pede para o usuário digitar o CPF e verifica se ele é válido. Para isso, primeiramente o programa deve multiplicar cada um dos 9 primeiros dígitos do CPF pelos números de 10 a 2 e somar todas as respostas. O resultado deve ser multiplicado por 10 e dividido por 11. O resto dessa divisão deve ser igual ao primeiro dígito verificador (10º dígito). Em seguida, o programa deve multiplicar cada um dos 10 primeiros dígitos do CPF pelos números de 11 a 2 e repetir o procedimento anterior para verificar o segundo dígito verificador.

Exemplo:

Se o CPF for 286.255.878-87 o programa deve fazer primeiro:

x = (2\*10 + 8\*9 + 6\*8 + 2\*7 + 5\*6 + 5\*5 + 8\*4 + 7\*3 + 8\*2)

Em seguida, o programa deve testar se x\*10%11 == 8 (o décimo número do CPF). Se sim, o programa deve calcular:

x = (2\*11 + 8\*10 + 6\*9 + 2\*8 + 5\*7 + 5\*6 + 8\*5 + 7\*4 + 8\*3 + 8\*2)

e verificar se x\*10%11 == 7 (o décimo primeiro número do CPF).

# Funções

### 1

Faça uma função que recebe um número e imprime seu dobro.

### 2

Faça uma função que recebe o valor do raio de um círculo e retorna o valor do comprimento de sua circunferência: C = 2pir.

### 3

Faça uma função para cada operação matemática básica (soma, subtração, multiplicação e divisão). As funções devem receber dois números e retornar o resultado da operação.

### 4

Faça uma função que recebe um nome e imprime “olá, [nome]”.

### 5

Faça uma função que recebe um nome e um horário e imprime “Bom dia, [nome]”, caso seja antes de 12h, “Boa Tarde, [nome]”, caso seja entre 12h e 18h e “Boa noite, [nome]” se for após às 18h.

### 6

Faça uma função que recebe um número e retorna true se ele é par ou false, se ele é ímpar.

### 7

Faça uma função que sorteia 10 números aleatórios entre 0 e 100 e retorna o maior entre eles.

### 8

Faça uma função que recebe um número n de entrada, sorteia n números aleatórios entre 0 e 100 e retorna a média deles.

### 9

Faça uma função que recebe um vetor de palavras e retorna um vetor contendo as mesmas palavras do vetor anterior, porém escritas em caixa alta.

### 10

Faça uma função que recebe dois vetores e retorna a soma item a item desses vetores.

Exemplo: Se a função receber os vetores [1,4,3] e [3,5,1], então a função deve retornar [1+3, 4+5, 3+1] = [4, 9, 4].

### 11

Faça uma função que receba dois vetores e retorne o produto item a item desses vetores.

Exemplo: Se a função receber os vetores [1,4,3] e [3,5,1], então a função deve retornar [1x3, 4x5, 3x1] = [3, 20, 3].

### 12

Faça uma função que recebe um número x e um vetor numérica e retorna um vetor cujos elementos são os itens do vetor de entrada multiplicado por x.

Exemplo:

Se a função receber o número 5 e o vetor [3,5,1], então a função deve retornar [5x3, 5x5, 5x1] = [15, 25, 5].

### 13

Faça uma função que recebe um vetor de números e retorna a soma dos elementos desse vetor.

### 14

Faça uma função que recebe um vetor de números e retorna a média aritmética dos elementos desse vetor.

### 15

Faça uma função que recebe um número e retorna o número invertido.

Exemplo x = 32243;  
Saída esperada: 34223

### 16

Faça uma função que recebe uma string como parâmetro e converte a primeira letra de cada palavra para maiúsculo.

Exemplo: ‘the quick brown fox’  
Saída esperada: ‘The Quick Brown Fox’

### 17

Faça uma função que recebe um número e retorna um booleano representando se ele é primo ou não.

### 18

Faça uma função que recebe um argumento e retorna o seu tipo de dado (number, string, etc).

### 19

Faça um função que recebe um vetor de números e encontre o segundo menor e o segundo maior número, respectivamente.

Exemplo: [1,2,3,4,5]  
Saída esperada: 2,4

### 20

Faça uma função que recebe um vetor numérico e um número e retorne um vetor com os elementos de maiores que esse número.

### Desafio 1

Faça uma função que receba um número e calcule seu fatorial.

### Super Desafio!

Repita o exercício anterior usando recursão, ou seja, uma função que chame ela mesma, lembrando que 3! = 3\*2!, que 2! = 2\*1!, que 1! = 1\*0! e que 0! = 1.

### Desafio 2

Faça uma função que recebe duas entradas: um input dado pelo usuário e um string que informa o tipo de dado (“idade”, “salário” ou “sexo”), e verifica se os dados digitados foram válidos, usando os seguintes critérios:

a. Idade: entre 0 e 150;

b. Salário: maior que 0;

c. Sexo: M, F ou Outro.

### Desafio 3

A sequência Fibonacci é a sequência cujos dois primeiros termos são 1 e os demais são obtidos através da soma de seus dois antecessores, isso é:

a. Fibonacci(1) = 1 e Fibonacci(2) = 1;

b. dado qualquer número n >= 3, Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)

Assim, os 10 primeiros termos da sequência Fibonacci são:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55…

Faça uma função que receba um número n e calcule o termo de número n da sequência Fibonacci.

### Super Desafio!

Refaça o desafio 3 usando recursão.