Introdução ao JavaScript

JavaScript é uma linguagem de sintaxe semelhante ao Java que foi concebida para web. Ela roda dentro dos navegadores para permitir conteúdos dinâmicos em sites.

Hoje, JavaScript extrapolou o navegador e se tornou uma linguagem tanto de lado cliente como de lado servidor por meio do **nodeJS**.

É uma linguagem extremamente popular, pois por anos foi a única opção para desenvolvimento frontend. Hoje temos linguagens que são transpiladas como TypeScript e outras opções que ainda não tiveram muita adesão por meio de WebAssembly, no entanto, JavaScript continua dominante.

Ela é uma linguagem de tipagem dinâmica e fraca, o que permite grande flexibilidade e simplicidade ao custo de comprometer um pouco a clareza e manutenibilidade do código.

JavaScript trata funções como cidadãos de primeira classe, o que quer dizer que funções são valores, o que é um pré-requisito para fazer programação no estilo funcional.

Javascript também permite o uso da orientação a objetos, ainda que não seja implementado com o rigor de linguagens como Java, C++ ou C#.

Hoje, entre navegador, servidor (**node**) e mobile (**React Native**) o JavaScript faz parte talvez do único stack (com adesão) capaz de fazer frontend, backend e mobile usando a mesma linguagem.

Sua capacidade comportar códigos estruturados, orientados a objetos e funcionais na mesma linguagem é bastante atrativo para vários nichos de programadores.

Setup

Executando no Navegador

A forma mais simples de executar um cógido em JavaScript é usando o navegador. Para isso, será necessário criar um arquivor html e referenciar o seu código em JavaScript por lá. Primeiro, crie um arquivo *index.html* básico:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Título</title>
</head>
<body>
</body>
<script src="script.js"></script>
</html>
```

A tag importante para executar o código é a tag *script*. Onde no atributo *src* colocamos o endereço para o nosso arquivo em JavaScript. Para tanto, crie na mesma pasta um arquivo *script.js* com o seu código. Para testar adicione ao arquivo:

```
alert("Funciona!");
```

Em seguida abra o arquivo *index.html* com algum navegador, que deve aparecer uma janela alertando o texto passando em JavaScript "Funciona!".

Executando fora do Navegador

Por conveniência, enquanto estamos apenas estudando a linguagem há a opção de executar o script fora do navegador usando o **NodeJS**. Para isso, você deve instalar no seu computador o node.

Instalando no Windows

Baixe o instalador diretamente de **https://nodejs.org/**

Instalando no Mac

Usando o Homebrew, basta abrir o terminal e executar o comando:

```
brew install node
```

Para verificar se a instalação foi bem sucedida, faça `node -v`, para verificar a versão instalada.

Instalando no Linux

Consulte em **https://nodejs.org/en/download/package-manager/** o procedimento para cada distribuição Linux.

Após ter o node instalado, basta abrir a linha de comando e ir até o diretório e executar:

```
node [nome_do_arquivo]
```

No nosso caso:

```
node script.js
```

Note que por estar fora do navegador alguns comandos em JavaScript não funcionam, como é o caso do *alert*. Teste imprimindo no *console*:

```
console.log("Funciona!");
```

Note que executando o `console.log("Funciona!");` no navegador, o texto será impresso no console do navegador, dentro do *Developer tools*.

Variáveis

Variáveis são o conceito mais básico da programação. De forma super simplificada é atribuir um "apelido" a um valor. Isso permite que esse valor seja lido e modificado durante o programa.

No entanto, por baixo dos panos, seu programa está solicitando que o sistema operacional da sua máquina reserve um pedaço de memória RAM para guardar esse valor. Diferentes tipos de valor têm tamanhos diferentes. Por isso, o sistema precisa saber de antemão qual tipo utilizar.

Existem duas vertentes de linguagens que tratam isso de forma diferente:

Uma, a **estática**, obriga a variável ser de um tipo exclusivo de informação. Isto é, uma variável numérica é sempre numérica. Você pode alterar o número armazenado mas não pode mudar a informação para texto por exemplo. Geralmente essas linguagens exigem declarar o tipo ao criar a variável para reservar memória, mesmo que a variável ainda não tenha valor.

A segunda, a **dinâmica**, em que o tipo é inferido implicitamente e este pode ser mudado ao longo do código. A memória só é alocada se a variável tiver um valor. Este é o caso de JavaScript.

O tipos de informação que podem ser salvos em variáveis são:

Primitivos:

- undefined -> Não há tipo nem valor
- boolean -> verdadeiro ou falso
- string -> textos
- number -> números

Complexos:

- function -> funções
- object -> objetos

Tipos primitivos são as informações básicas que uma linguagem consegue armazenar. Todas construção de dados mais complexos decorre da utilização desses. Também podem ser armazenados funções (muito importante) e objetos em variáveis. Os tipos primitivos são, números, texto (representado entre aspas) e booleano (valor lógico verdadeiro ou falso).
O *undefined* representa uma variável sem valor atribuído.

Podemos fazer uma variável de qualquer um dos tipos acima.

```
//boolean
var a = true;
var b = false;

//Number
var c = 10;
var d = 11.5;

//String
var e = "teste";

//Undefined
var f;
var g = undefined;
//function
var a = function(){};
var b = () => {};

//object
var c = {};
var d = []; //array
var e = null;
```

Aqui usamos *var*, mas vamos padronizar a partir de agora a palavra *let* para declaração de variáveis e a palavra *const* para declaração de constantes. O *null* representa uma informação vazia. Diferente de *undefined*, ele mostra que a variável foi declarada mas com valor vazio.

Em JavaScript, temos um comportamento estranho das variáveis. Elas podem ser declaradas depois de utilizadas, pois elas são levantadas para o início do escopo. Isso se chama hoisting.

Usando *var* variáveis não respeitam seu escopo, uma variável declarada em uma função estará disponível depois do escopo da função.

Esses comportamentos são peculiares e diferentes de muitas linguagens, por isso padronizamos usar *let* em vez de *var*, o que fará com que os escopos sejam respeitados.

O *const* impede que o valor possa ser alterado após ser iniciado, gerando um erro ao ser tentado.

```
const pi = 3.1415;
```

Condicionais

Operadores Lógicos

Em programação, condicionais são esturtura de decisões. O código executo de uma maneiro ou de outra a depender de uma condição, que por sua vez será interpretada como verdadeiro ou falso. Esse tipo de dado é chamado de *booleano* e possui valor *true* ou *false*. Uma condição é uma operação lógica que tem como resultado um valor booleano. Os operadores de comparação em JavaScript são:

Operador	Nome	Exemplo	Exemplo
>	Maior	2 > 0	true
>=	Maior ou igual	2 >= 2	true
<	Menor	2 < 0	false
<=	Menor ou igual	2 <= 1	false
==	Igual	1 == '1'	true
===	Igual em valor e tipo	1 === '1'	false
!=	Diferente	1 != 2	true
!==	Diferente em valor ou tipo	'5' !== 5	false

Os operadores lógicos em JavaScript são:

Operador	Nome	Exemplo	Exemplo
&&	E	2 > 0 && 1!==1	false
||	Ou	2 > 0 || 1!==1	true
!	Não	!(1===1)	false

Os operadores de comparação retornam um booleano a depender do resultado da comparação. E os operados lógicos fazem operações sobre valores booleano. O operador && só retorna verdadeiro se as duas condições forem verdadeira. Enquanto para o operador || basta uma das condições ser para o resultado ser verdadeiro. O operador ! inverte o valor lógico, ou seja, verdadeiro vira falso e vice-versa. Confira as tabelas verdades:

A	B	A&&B	A||B
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

If/Else

A estrutura condicional em JavaScript é da seguinte maneira:

```
let condicao = x > 0;

if(condicao){
    console.log("X é maior do que zero");
}

else{
    console.log("X é menor ou igual a que zero");
}
```

A condição deve estar entre parêntesis. Para fim de ilustração o resultado condição foi guardada em uma variável (linha 1), porém é prática escrever diretamente dentro dos parêntesis. Caso a condição seja *true* o código dentro do *if* é executado, senão o do *else* é executado.

Ainda é possível fazer estruturas que verifiquem mais de uma condição usando o *else if*:

```
if(x > 0){
    console.log("X é positivo");
}

else if(x == 0){
    console.log("X é zero");
}

else{
    console.log("X é negativo");
}
```

Caso a condição do *if* for *false*, é verificado a condição do *else if*, se esta por sua vez for falsa é verificado a condição do próximo *else if*, se houver, senão é executado o *else*, também se houver.

Operador Tenário

Para atribuição condicional de valor o JavaScript possui um operador que permite fazer uma *if else* inline, isto é, em uma única linha. Operador tenário é representado por *?*:

```
let paridade = x % 2 === 0 ? 'par' : 'impar';
```

Primeiro colocamos a condição a ser verificada, no caso `x % 2 === 0` onde *%* é o operador que retorna o resto da divisão. Ou seja, a condição é se o resto da divisão é igual a zero. Se sim, guardamos a string 'par' na variável, se não, 'impar'. Note que os valores são separados por *:* (dois pontos).

Switch

O Switch é uma estrutura condicional que recebe um valor e executa um código quebrando os casos que a variável pode receber. Por exemplo:

```
let hoje = new Date().getDay();
let dia;

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

O comando `new Date().getDay()` retorna o dia atual como um inteiro, começando em 0 no domingo e indo até 6 para o sábado. O *switch* recebe a variável e quebra casos. Caso seja 1, por exemplo, definimos a variável *dia* como "Segunda". O *break* representa o fim do case. Caso seja omitido o computador executará o caso seguinte. Por isso é dispensável no último caso. Ainda é possível definir um padrão caso nenhum dos casos exista usando o *default*.

```
let sinal = 'azul';
switch(sinal){
    case 'verde' :
        console.log('pode passar');
        break;
    case 'amarelo' :
        console.log('prestes a fechar, cuidado...');
        break;
    case 'vermelho':
        console.log('fechado, não passe');
        break;
    default:
        console.log('esse não é um valor válido');
        break;
}
```

Truthy e Falsy

Além do *true* e do *false*, o JavaScript aceita outras informações que não são booleanos e os interpreta como se fosse *true* ou *false*. Esses casos chamamos de *truthy* e *falsy*. Por exemplo, o JavaScript interpreta os seguintes valores como falso:

- 0
- '' ou ""
- null
- undefined
- NaN

Todos os demais são interpretados como verdadeiro. Alguns exemplos de *truthy*:

- []
- {}
- function(){}

Uma utilidade dessa característica é verificar se uma variável está definida antes de usá-la. Assim evitando erro.

Laços de Repetição

Afim de fazer operações repetitivas, o JavaScript oferece algumas opções de estruturas de repetições. Também chamados de laços ou *loops*.

While

O *while* é a estrutura que executa um código enquanto uma condição for verdadeira. Exemplo:

```
let count = 0;
while(count < 100){
    console.log(count);
    count++;
}
```

A condição dada para a execução é *count* menor do que 100. Assim enquanto essa condição retornar *true*, o código será executado. Na linha 4 há a atualização da variável *count*, o ++ representa a operação em aumentar em 1 o valor da variável. Note que se não houvesse essa linha a condição nunca se tornaria falsa, uma vez que continuaria tendo valor 0. Neste caso o laço seria infinito. Preste sempre atenção para evitar esse tipo de cenário. Note também que se a condição for falsa logo de início, por exemplo se mudarmos a linha 1 para `let count = 200;`, o código dentro do *while* nunca seria executado.

Do-While

Há casos em que é necessário executar o *while* pelo menos uma vez. Para isso temos a estrutura *do-while*, onde primeiro se executa o código e depois é verificado a condição para continuar executando ou não. O mesmo exemplo:

```
let count = 0;
do{
    console.log(count);
    count++;
}
while(count < 100);
```

For

O *for* é uma estrutura de repetição que é executado um número específico de vezes. Nele é declarado um variável com um valor inicial, depois é determinado a condição de parada e por fim a passo dado entre o valor inicial e o final. Exemplo:

```
for(let i = 0; i < 10; i++){
    console.log(i+1);
}
```

A variável declarada é o *i*, onde esta é iniciada com valor 0. A condição de parada é ao *i* atingir valor 10, demonstrado em `i < 10`. Por fim, o `i++` mostra que o *i* vai de 0 a 9 de em incrementos de um. Note que a variável declarado no *for* pode possuir qualquer nome.

Vetores

Vetor é uma sequência ordenada de valores. Também é chamado pelo nome em inglês, *array*. É denotado pelo uso de colchetes, com os valores separados por vírgula. Exemplo:

```
let vetor = [1,2,3,4,5,6,7,8,9,10];
```

Também é possível declarar o vetor vazio e ir adicionando pela posição os elementos:

```
let vetor = [];
vetor[0] = 'teste';
vetor[1] = 'teste2';
```

Além de índice numérico é possível fazer índice associativo usando strings.

```
vetor['indice1'] = 'teste';
vetor['indice2'] = 'teste2';
```

Vetores são heterogêneos, isto é, podem guardar valores de tipos diferentes:

```
let vetor = [1,2,3,'a','b','c',true,false];
let primeiro_elemento = vetor[0];
let quinto_elemento = vetor[4];
```

Porém, uma boa prática é construir vetores de um único tipo de dado. Para conseguir acessar um elemento específico, basta passar em colchetes o índice (posição) do elemento no vetor. Sendo a contagem da esquerda para a direita, iniciando em 0.

Percorrendo o Vetor

Usando a estrutura de repetição *for* é possível percorrer todos os elementos de um vetor:

```
let vetor = [1,2,3,4,5,6,7,8,9,10];

for(let i = 0; i < vetor.length; i++){
    console.log(vetor[i]);
}
```

O *i* do *for* vai do valor 0 até o comprimento do vetor menos um. Note que `vetor.length` retorna o comprimento do mesmo.

Há ainda dois laços *for* especiais que permitem percorrer um vetor com maior facilidade. O *for-of*:

```
const vetor = [10,20,30,40,50];
for(let valor of vetor){
    console.log(valor);
}
```

O *for-of* recebe um vetor e a variável declarada vai possuir os valores dos elementos do vetor. Assim percorrendo diretamente o array inteiro.

A alternativa é o *for-in*, onde a variável declarada assume o valor dos índices do vetor passado:

```
const vetor = [10,20,30,40,50];

for(let indice in vetor){
    console.log(indice, vetor[indice]);
}
```

Matrizes

Matrizes são vetores muiltidimencionais, isto é, é um vetor de vetores. Portanto tudo que se aplica a vetor também vale para matrizes. A declaração de uma matriz segue o padrão

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

for(let linha of matriz){
    for (let dado of linha){
        console.log(dado);
    }
}
```

Para percorrer os elementos de uma matriz é necessário aninhar laços *for*. O primeiro para percorrer as linhas e o segundo os elementos dessa linha.

Vetores Dinâmicos

Em JavaScript vetores são estruturas de dados dinâmicas, isto é, de tamanho variável. Para adicionar um elemento ao final do vetor, faça:

```
let vetor = [10,20,30,40,50];
let novo_elemento = 60;

vetor.push(novo_elemento);
```

Para remover do final:

```
let vetor = [10,20,30,40,50];
vetor.pop();
```

Para adicionar ao começo:

```
let vetor = [10,20,30,40,50];
let novo_elemento = 0;

vetor.unshift(novo_elemento);
```

Para remover do começo:

```
let vetor = [10,20,30,40,50];
vetor.shift();
```

Funções

Podemos ter duas interpretações do que são funções. A primeira de que uma função é um trecho de código em que damos um nome e que é executado a cada vez que é chamado. Exemplo:

```
// Definindo a função
function geradorNumerico(){
    for(let i = 0; i < 10; i++){
        console.log(i);
    }
}

// Chamando a função
geradorNumerico();
```

Criamos um função usando o `function` e demos o nome `geradorNumerico`. É uma função que apenas imprime no console os número de 0 a 100. Cada vez que é chamada a função, o seu código é executado. Para tornar a função mais reutilizável, podemos parametrizá-la. Para isso, adicionamos os parâmetros na declaração.

```
// Definindo a função
function geradorNumerico(comeco, fim){
    for(let i = comeco; i <= fim; i++){
        console.log(i);
    }
}

// Chamando a função
geradorNumerico(10,20);
```

Entre os parêntesis adicionamos a declaração de duas, variáveis `comeco` e `fim`. De forma que a função imprime os números entre esses dois valores.

A segunda interpretação de função é a matemática, como algo que recebe valores e devolve outro.

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

const resultado = soma(2,3);
```

Declaramos dois parâmetros, *a* e *b*, que ao ser passados a função *soma*, esta retorna a soma dos dois. Esse valor fica disponível para ser salvo em outra variável ou para ser passado como valor em outra função.

Formas de Declarar

Além da forma tradicional de declarar, há ainda duas formas de utilizar funções, que é usando do artifício de que em JavaScript funções são valores.

```
// Usando function
function soma1(a,b) { return a+b; }
// Atribuindo uma função anônima em constantes
const soma2 = function(a,b){ return a+b; };
// Atribuindo uma função de "flecha" em constantes
const soma3 = (a,b) => a+b;
```

A primeira função foi declarada como a forma tradicional apresentada antes. No segundo caso foi usado uma função anônima, isto é, sem nome. Note que seguido de `function` não há declaração de um nome para a função. O que define o seu nome é a variável. No terceiro caso é também uma função anônima porém é usada a chamada função de flecha. Onde é declarado os parâmetros de entrada em parêntesis e a sua definição seguido da flecha `=>`.

Exemplos de funções de flecha:

```
const hello1 = () => "Hello World!";
const hello2 = (name) => "Hello " + name;
const hello3 = (name) => {return "Hello " + name;};
```

Note que não é possível fazer qualquer restrição de tipo de entrada como parâmetro de uma função. Caso seja passado algo que não era esperado, possivelmente quebrará o código. Fique atento.

Funções de Alta Ordem

Em JavaScript, funções são cidadãos de primeira classe. O que significa que podem ser tratados como valores e salvo em variável. Portanto, naturalmente funções também podem ser passadas como parâmetro para uma função. Em resumo, uma função de alta ordem é uma função que recebe ou retorna uma função.

Recebendo uma Função como Parâmetro

```
const somar = (a, b) => a + b;
const subtrair = (a, b) => a - b;
const aplicarOperacao = (a, b, operacao) => operacao(a,b);

const resultado1  = aplicarOperacao(1,2,somar);
const resultado2 = aplicarOperacao(1,2,subtrair);
```

No código acima declaramos três funções, *somar*, *subtrair* e *aplicarOperacao*. As duas primeiras recebem dois parâmetros e devolvem um valor. A *aplicarOperacao* recebe três parâmetros, sendo o terceiro uma função a ser aplicada passando os dois primeiros parâmetros. Nas linhas 5 e 6 são passados, respectivamente, as funções *somar* e *subtrair*.

Retornando Funções

Nós podemos retornar funções como uma forma de construir abstrações mais complexas, onde uma função é um geradora de funções. No exemplo a seguir, temos a função *somarX*, que recebe *x* de parâmetro e retorna uma função que recebe *y* e retorna *x+y*. Note que ao passar 1 para *somarX*, criamos uma função que soma 1. E assim sucessivamente.

```
const somarX = (x) => (y) => x + y;

const somar1 = somarX(1);
const somar2 = somarX(2);
const somar3 = somarX(3);
```

Nas próximas três aulas vamos estudar funções de alta ordem para vetores.

Map

A função Map é o primeiro caso de aplicação de função de alta ordem que vamos ver. Essa função é usada para transformar vetores. Passamos uma função para o Map, e essa função é aplicada a cada item do vetor.

Vamos supor que temos um vetor numérico e queremos criar um vetor que é o dobro do anterior.

```
const vetor = [1,2,3,4,5,6];

const dobro = (item)=>2*item;
```

```
const vetorDobrado = vetor.map(dobro);
```

Criamos a variável *dobro* para salvar a função que recebe um item e retorna o mesmo multiplicado por dois. Ao passar como parâmetro para o *map*, que é utilizado com um ponto na variável que armazena o nosso vetor, a função dobro é chamado onde cada item do vetor é passado de parâmetro. Então, o resultado é salvo numa nova variável *vetorDobrado*, isso é necessário pois o *map* não altera o vetor original.

É possível passar mais dois parâmetros para as funções no *map*. O primeiro parâmetro é sempre o próprio elemento, o segundo é a sua posição no vetor e o terceiro é o próprio vetor.

```
// Função que eleva ao quadrado
const aoQuadrado = (item, indice, vetor) => vetor[indice]*item;
```

É possível escrever a função diretamente dentro do *map*.

```
const vetor = [1,2,3,4,5,6];

const vetorTransformado = vetor.map((x)=>x+1);
```

O *map* exige que seja passado pelo menos um parâmetro para a função. Então para usar métodos de um tipo específico de dado também é necessário fazer a declaração de uma função. Exemplo:

```
const vetor = ["a","b","c"];
const toUpper = (str) => str.toUpperCase();
const maiusculas = vetor.map(toUpper);
```

No código acima tínhamos um vetor de caractéres que para passar para maiúsculo definimos a função *toUpper*, que recebe uma string e aplica o método *toUpperCase*.

Filter

Filter é uma função de alta ordem semelhante ao *map*, a diferença é que o objetivo do *filter* é filtrar elementos do vetor. Portanto a função passada para o deve receber o elemento e retornar um booleano. Se retorna *true* o elementor será mantido, senão retirado. Como exemplo, suponha que temos um vetor numérico e queremos somente os pares:

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

const pares = vetor.filter(x => x % 2 === 0);
```

Note que estamos passando a função `x => x % 2 === 0` para o filter. Ele recebe um elemento x do vetor e retorna *true* se o resto da divisão por dois é zero.

Um outro exemplo é retirar elementos maior do que um valor limite:

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

const vetor2 = vetor.filter(x => x < 8);
```

No exemplo acima filtramos os elementos maiores ou iguais a oito.

Reduce

O objetivo *reduce* é reduzir um vetor a um valor ou objeto. Por exemplo, somar todos os elementos de um vetor é reduzir ele a um valor. O *reduce* é um pouco mais complexos que o *map* e o *filter* por que deve ser passado um parâmetro a mais. Vejamos o exemplo da soma:

```
const vetor = [1,2,3,4,5,6];

const soma = vetor.reduce((estado, item) => estado + item);
```

Além do elemento do vetor, é necessário passar a variável que vai armazenar a evolução do estado ao longo da aplicação da função no vetor. No caso podemos pensar nessa variável *estado* como um acumulador que guarda a soma parcial até o presente elemento da iteração. Assim, a função recebe a soma acumulada e o novo item, retornando o estado somado ao item. Esse valor então é passado como estado para o elemento seguinte. Ao percorrer todo o vetor, o valor dessa variável estado é retornado. O primeiro elemento não recebe um estado por ser o primeiro, então o segundo elemento recebe o primero como estado. Porém, é possível declarar explicitamente qual seria o estado inicial a ser passado para o primeiro elemento, basta passar como parâmetro para o *reduce*:

```
const vetor = [1,2,3,4,5,6];

const soma = vetor.reduce((estado, item) => estado + item, 0);
```

Na linha 3 adicionamos o zero como estado inicial. Em casos simples como esse não será necessário declarar explicitamente, mas ao trabalhar com objetos é necessário.

Vamos fazer um exemplo com objetos, caso ainda não tenha familiaridade ainda retorne para reler esse trecho após o módulo de introdução a orientação a objeto.

Suponha que tenha um vetor de objetos aluno que possuem três atributos: nome, nota 1 e nota 2.

```
let vetor = [
  { nome : 'nome1', nota1 : 5, nota2 : 4 },
  { nome : 'nome2', nota1 : 4, nota2 : 3 },
  { nome : 'nome3', nota1 : 7, nota2 : 8 },
  { nome : 'nome4', nota1 : 2, nota2 : 7 },
  { nome : 'nome5', nota1 : 9, nota2 : 9 },
];
```

Vamos usar o *reduce* para somar todas a notas 1 e 2 dos alunos. Para isso, declaramos primeiro o objeto a ser recebido como estado inicial:

```
const estadoInicial = {
  somaNota1 : 0,
  somaNota2 : 0,
  qtdNota1 : 0,
  qtdNota2 : 0,
};
```

Em seguida passamos como parâmetro para o reduce:

```
const result = vetor.reduce((estado, valor) => {
  return {
      somaNota1 : estado.somaNota1 + valor.nota1,
      somaNota2 : estado.somaNota2 + valor.nota2,
      qtdNota1 : estado.qtdNota1 + 1,
      qtdNota2 : estado.qtdNota2 + 1
  };
```

```
}, estadoInicial);
```

Note que podemos escrever em linhas separadas para facilitar a escrita e leitura. Esse reduce, a cada rodada, cria um novo objeto que contém a somatória das notas do estado anterior com o valor das notas do item atual. Também contém um contador para cada nota para simplificar no cálculo de uma média posteriormente. Observe que o estado inicial foi passado por parâmetro depois da função no reduce. Sem ele não teríamos de onde tirar os valores que estão no objeto.