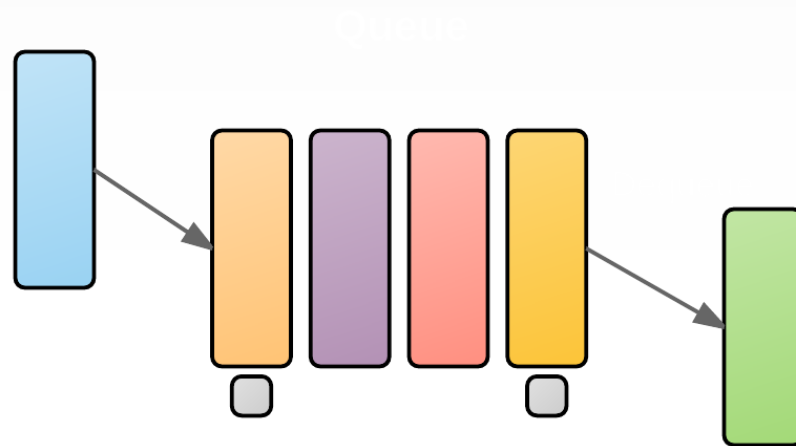


JavaScript

Módulo Básico





VETORES

- **Estrutura de Dados: Vetores**
 - **Vetores** ou **arrays** são estruturas que permitem armazenar uma lista de dados na memória principal do computador
 - Úteis para **inserir** ou **remover** itens de uma lista de compras ou de alunos de uma turma, permitindo **recuperar/manipular** todos os itens inseridos na lista
 - Um índice numérico (**começa em 0**) identifica cada elemento da lista

- ***Estrutura de Dados: Vetores***
 - ***Uma **lista** de **itens** de um supermercado armazenada no vetor produtos***

Tabela 1 – Representação dos itens/elementos de um vetor

Produtos	
0	Arroz
1	Feijão
2	logurte
3	Leite
4	Suco
5	Pão

- **Estrutura de Dados: Vetores**

- Para *referenciar um item do vetor*, devemos *indicar seu nome*, seguido por um *número* entre colchetes que aponta para o seu *índice*

- É importante reforçar que o *vetor inicia pelo índice 0* (zero)

- Para obter o primeiro produto inserido no vetor, utilizamos:

`produtos[0]`

- Alterando um produto da lista, com uma nova atribuição de conteúdo a um elemento do vetor

`produtos[2] = "Queijo"`

- **Estrutura de Dados: Vetores**

- Na linguagem JS, **não** é necessário indicar o número total de elementos do **vetor** na sua declaração
- Para declarar um **vetor** em JS, devemos **utilizar** uma das seguintes formas:

```
const produtos = []  
const produtos = new array()
```

- É possível **declarar** um **vetor** com algum conteúdo inicial (e, mesmo assim, **adicionar** ou **remover** itens no vetor)

```
const produtos = ["Arroz", "Feijão", "Iogurte"]
```

- **Estrutura de Dados: Vetores**
 - **Vetores** podem ser **declarados** com **const** e mesmo assim ter o **valor** dos **seus elementos** alterados
 - O que **não** pode ser **feito** com o **const** é uma **reatribuição** de valor a uma **variável**
 - Isso evita possíveis **erros** que poderiam ocorrer com o **var**, principalmente em **programas maiores**



- ***Estrutura de Dados: Vetores***

- ***Diferença entre o uso de variáveis e vetores:***

- ***Uma variável armazena apenas **um valor por vez**; quando uma nova atribuição a essa variável é realizada, o seu valor anterior é perdido***
 - ***Após as duas atribuições, a variável **idade** permanece apenas com o último valor que lhe foi atribuído***

let idade

idade = 18

idade = 15

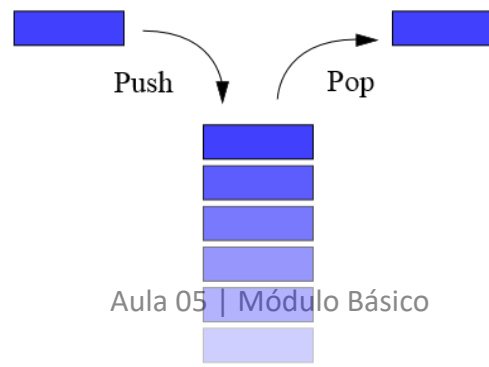
- ***Estrutura de Dados: Vetores***

- ***Diferença entre o uso de variáveis e vetores:***

- ***Os vetores permitem armazenar um conjunto de dados e acessar todos os seus elementos pela referência ao índice que identifica cada um deles***
 - ***Assim, após as duas atribuições, os dois valores atribuídos ao vetor *idade* podem ser acessados***

```
const idade = []  
idade[0] = 18  
idade[1] = 15
```

- *Inclusão e exclusão de itens*
 - Métodos de *inclusão* e *exclusão* de itens em vetores
 - *push()*: **adiciona** um elemento ao **final** do vetor
 - *unshift()*: **adiciona** um elemento ao **início** do vetor e **desloca** os elementos existentes uma **posição abaixo**
 - *pop()*: **remove** o **último** elemento do vetor
 - *shift()*: **remove** o **primeiro** elemento do vetor e **desloca** os elementos existentes uma **posição acima**



- *Inclusão e exclusão de itens*

- *Exemplos:*

```
<script>
```

```
  const cidades = ["Pelotas"]           // declara e define conteúdo inicial do vetor
```

```
  cidades.push("São Lourenço")          // adiciona cidade ao final do vetor  
  console.log(cidades)                  // ['Pelotas', 'São Lourenço']
```

```
  cidades.unshift("Porto Alegre")       // adiciona ao início e desloca as demais  
  console.log(cidades)                  // ['Porto Alegre', 'Pelotas', 'São Lourenço']
```

```
  const ultima = cidades.pop()           // remove a última cidade do vetor  
  console.log(cidades)                  // ['Porto Alegre', 'Pelotas']
```

```
  const primeira = cidades.shift()       // remove a primeira e "sobe" as demais  
  console.log(cidades)                  // ['Pelotas']
```

```
</script>
```

- **Inclusão e exclusão de itens**
 - O método **splice** (**emendar**) pode possuir diversos parâmetros e ser utilizado para **alterar**, **inserir** ou **remover** elementos do array
 - O método **slice** (**fatiar**) obtém uma “fatia” de um vetor
 - Contém dois parâmetros que são posição **inicial** e **final** (**não obrigatória**) do array
 - Se posição **inicial** for um **número negativo**, ela indica a quantidade de elementos do **final** para o **início** que serão obtidos
 - Se posição **final** for um **número negativo**, ela indica a quantidade de elementos do **fim** para o **início** que devem ser descartados

- ***Inclusão e exclusão de itens***

- ***Exemplos:***

```
<script>
```

```
  const letras = ["A", "B", "C", "D"]
```

```
  const letras2 = letras.slice(-2)
```

```
  const letras3 = letras.slice(0, -1)
```

```
  console.log(letras)
```

```
  console.log(letras2)
```

```
  console.log(letras3)
```

```
  // declara e define conteúdo inicial do vetor
```

```
  // obtém 2 últimas letras
```

```
  // obtém do início até final, exceto a última
```

```
  // ['A', 'B', 'C', 'D']
```

```
  // ['C', 'D']
```

```
  // ['A', 'B', 'C']
```

```
  const retira = letras.splice(2, 1)
```

```
  console.log(letras)
```

```
  console.log(retira)
```

```
  // remove a partir da posição 2, 1 elemento
```

```
  // ['A', 'B', 'D']
```

```
  // ['C']
```

```
</script>
```

- **Inclusão e exclusão de itens**

- **Exemplos:**

```
<script>
```

```
const letras = ["A", "B", "C", "D"] // declara e define conteúdo inicial do vetor
```

```
const letras2 = letras.slice(-2) // obtém 2 últimas letras
```

`slice()` não modifica o conteúdo do vetor original, enquanto que `splice()` modifica

A variável *retirada* é sempre um *array*, mesmo com um ou zero elementos

```
const retirada = letras.splice(2, 1) // remove a partir da posição 2, 1 elemento
```

```
console.log(letras) // ['A', 'B', 'D']
```

```
console.log(retirada) // ['C']
```

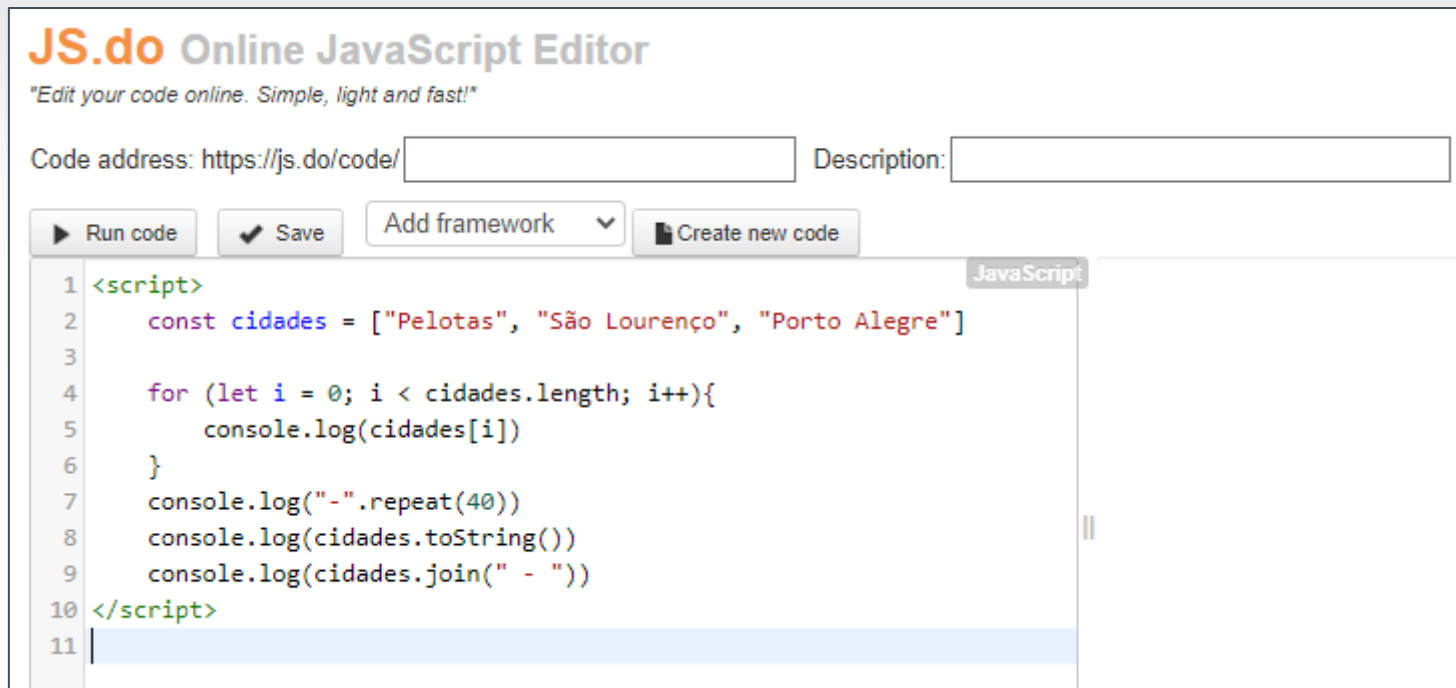
```
</script>
```

- **Tamanho do vetor e exibição dos itens**
 - A propriedade **length** retorna o número de elementos do vetor
 - Utilizamos essa propriedade quando for necessário **percorrer** a **lista**, realizar **exclusões** (**para verificar antes da exclusão, se a lista está vazia**) ou, então, para **exibir** o **número total de itens do vetor**
 - Para **percorrer** e **exibir** os elementos do **vetor cidades**, podemos utilizar o comando **for**, indicando que a **variável de controle i** começa em **0**, e **repetir o laço enquanto i** for **menor que cidades.length**

- **Tamanho do vetor e exibição dos itens**
 - **Outra forma de exibir o conteúdo do vetor é pelo uso dos métodos `toString()` e `join()`**
 - **Ambos convertem o conteúdo do vetor em uma `string`, sendo que no método `toString()` uma vírgula é inserida entre os elementos e no `join()` podemos indicar qual caractere será utilizado para separar os itens**



- ***Tamanho do vetor e exibição dos itens***
 - ***Exemplo (01):***
 - ***Formas de exibir o conteúdo de um vetor***

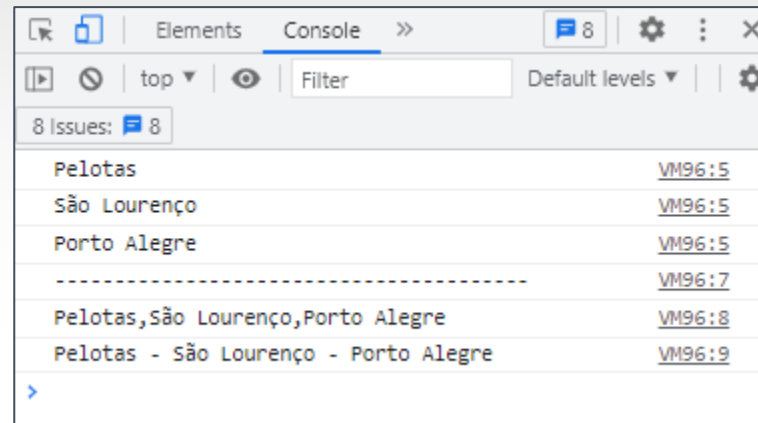


The screenshot shows the JS.do Online JavaScript Editor interface. At the top, it says "JS.do Online JavaScript Editor" with the tagline "Edit your code online. Simple, light and fast!". Below this, there are input fields for "Code address" (with a placeholder URL "https://js.do/code/") and "Description". A row of buttons includes "Run code", "Save", "Add framework" (with a dropdown arrow), and "Create new code". The main code editor area contains the following JavaScript code:

```
1 <script>
2   const cidades = ["Pelotas", "São Lourenço", "Porto Alegre"]
3
4   for (let i = 0; i < cidades.length; i++){
5     console.log(cidades[i])
6   }
7   console.log("-".repeat(40))
8   console.log(cidades.toString())
9   console.log(cidades.join(" - "))
10 </script>
11
```

Figura 1 – Manipulação de elementos do vetor

- ***Tamanho do vetor e exibição dos itens***
 - ***Exemplo (01):***
 - ***Formas de exibir o conteúdo de um vetor***



The screenshot shows a web browser's developer console with the 'Console' tab selected. It displays 8 issues, which are the elements of an array. The array contains the names of three cities: Pelotas, São Lourenço, and Porto Alegre. Each city name is followed by its index in the array, shown as a link (e.g., VM96:5). The array is displayed in a compact, readable format.

Pelotas	VM96:5
São Lourenço	VM96:5
Porto Alegre	VM96:5
-----	VM96:7
Pelotas, São Lourenço, Porto Alegre	VM96:8
Pelotas - São Lourenço - Porto Alegre	VM96:9

Figura 2 – Exibindo os elementos do vetor

- *For..of e forEach()*

- São equivalentes ao **for tradicional**, com uma sintaxe mais “**enxuta**”

```
for (const cidade of cidades) {  
    console.log(cidade)  
}
```

- A cada **iteração**, a variável **cidade** recebe um elemento do vetor **cidades**
- **Condições ou cálculos** podem ser realizados dentro do **loop** com essa variável

- *For..of e forEach()*

- São equivalentes ao **for tradicional**, com uma sintaxe mais “**enxuta**”

```
for (const cidade of cidades) {  
    console.log(cidade)  
}
```

- Como **cidade** é uma variável do bloco (**deixa de existir após cada iteração**), podemos **declará-la** como **const**
- A cada iteração a variável **não** é modificada, mas, sim, **deixa de existir e é novamente declarada**

- *For..of e forEach()*

- ***forEach()** é mais amplo e pode chamar uma função para manipular cada elemento do vetor*
- *Uma forma simples de utilizá-lo para percorrer os **elementos** de um vetor é apresentada a seguir*

```
idades.forEach((cidade, i) => {  
  console.log(`${i + 1}ª Cidade: ${cidade}`)  
})
```

- *Obtemos o conteúdo (**cidade**) e o índice (**i**) de cada elemento do vetor que estamos percorrendo – sendo que o índice (**i**) é opcional*

- *For..of e forEach()*

```
idades.forEach((cidade, i) => {  
  console.log(`${i + 1}ª Cidade: ${cidade}`)  
})
```

– *A saída desse fragmento de código (considerando o vetor **idades**)*

1ª Cidade: Pelotas

2ª Cidade: São Lourenço

3ª Cidade: Porto Alegre

- *For..of e forEach()*

- *Algumas operações sobre vetores mais simples se executadas a partir do **forEach**, como a soma dos elementos que compõem o vetor*

```
const numeros = [5, 10, 15, 20]  
let soma = 0
```

```
numeros.forEach(num => soma += num)  
console.log(`Soma dos Números: ${soma}`)
```

- *For..of e forEach()*
 - *Crie uma pasta **css**, **img** e **js***
 - *Crie um arquivo intitulado de **estilos.css** com o conteúdo abaixo, e salve dentro da pasta **css**:*

```
img.normal { float: left; height: 300px; width: 300px; }  
img.alta { float: left; height: 420px; width: 300px; }  
h1 { border-bottom-style: inset; }  
pre { font-size: 1.2em; }  
.fonteAzul { color: blue; }  
.oculta { display: none; }  
.exibe { display: inline; }  
.italico { font-style: italic; }
```


- ***For..of e forEach()***
 - ***Exemplo (02):***
 - ***Sistema odontológico de atendimento***

Consultório Odontológico

Paciente:

Em Atendimento: **Rafael**

1. Pedro
2. Patrícia
3. Bárbara
4. Rubens
5. Juliana



Figura 3 – Renderização do HTML do Exemplo (02)

- *For..of e forEach()*
 - *Exemplo (02):*

```
1  <!DOCTYPE html>
2  <html lang="pt-br">
3  <head>
4      <meta charset="UTF-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      <link rel="stylesheet" href="../css/estilos.css">
8      <title>Exemplo 01 | bkBank Academy</title>
9  </head>
10 <body>
11     
12     <h1> Consultório Odontológico </h1>
13     <form>
14         <p> Paciente:
15             <input type="text" id="inPaciente" required autofocus>
16             <input type="submit" value="Adicionar">
17             <input type="button" value="Urgência" id="btUrgencia">
18             <input type="button" value="Atender" id="btAtender">
19         </p>
20     </form>
21     <h3> Em Atendimento:
22         <span class="fonteAzul"></span>
23     </h3>
24     <pre></pre>
25     <script src="../js/exemplo_1.js"></script>
26 </body>
27 </html>
```

- ***For..of e forEach()***

- ***Exemplo (02):***

```
1 <!DOCTYPE html>
2 <html lang="pt-br">
```

O campo de entrada de dados contém um atributo **autofocus**. Define-se o campo do formulário em que o cursor ficará inicialmente posicionado.

Foi adicionado a tag **span**, dentro do h3. Com a tag **span**, pode-se identificar um local da página, que terá o seu conteúdo alterado pelo programa JS, sem que ocorra uma quebra de linha.

```
17 <input type="button" value="Gerar" id="btGerar">
18 <input type="button" value="Atender" id="btAtender">
19 </p>
20 </form>
21 <h3> Em Atendimento:
22 <span class="fonteAzul"></span>
23 </h3>
24 <pre></pre>
25 <script src="../../js/exemplo_1.js"></script>
26 </body>
27 </html>
```

- *For..of e forEach()*

- *Exemplo (02):*

```
1 <!DOCTYPE html>
2 <html lang="pt-br">
3 <head>
4   <meta charset="UTF-8">
5   <meta http-equiv="X-UA-Compatible" content="IE=edge">
6   <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

O nome do paciente será exibido na tag **h3** ao lado do texto "Em Atendimento: ". Foram acrescentados 2 botões no formulário, em acréscimos ao **submit** do form. O **evento click** será o responsável por disparar a programação associada a eles

```
17   <input type="button" value="Urgencia" id="btUrgencia">
18   <input type="button" value="Atender" id="btAtender">
19   </p>
20 </form>
21 <h3> Em Atendimento:
22   <span class="fonteAzul"></span>
23 </h3>
24 <pre></pre>
25 <script src="../../js/exemplo_1.js"></script>
26 </body>
27 </html>
```

- *For..of e forEach()*
 - *Exemplo (02): - Parte 1*

```
js > JS exemplo_1.js > ...
1  const frm = document.querySelector("form") // obtém elementos da página
2  const respNome = document.querySelector("span")
3  const respLista = document.querySelector("pre")
4
5  const pacientes = [] // declara vetor global
6
7  frm.addEventListener("submit", (e) => {
8    e.preventDefault()           // evita envio do form
9    const nome = frm.inPaciente.value // obtém nome do paciente
10   pacientes.push(nome)           // adiciona o nome no final do vetor
11   let lista = ""                  // string para concatenar pacientes
12
13   // for "tradicional" (inicia em 0, enquanto menor que tamanho do array)
14   for (let i = 0; i < pacientes.length; i++) {
15     lista += `${i + 1}. ${pacientes[i]}\n`
16   }
17
18   respLista.innerText = lista // exibe a lista de pacientes na página
19   frm.inPaciente.value = ""   // limpa conteúdo do campo de formulário
20   frm.inPaciente.focus()      // posiciona o cursor no campo
21 }
```

- *For..of e forEach()*
 - *Exemplo (02): - Parte 2*

```
23 // Adiciona um "ouvinte" para o evento click no btUrgencia que está no form
24 frm.btUrgencia.addEventListener("click", () => {
25     // verifica se as validações do form estão ok (no caso, paciente is required)
26     if (!frm.checkValidity()) {
27         alert("Informe o nome do paciente a ser atendido em caráter de urgência")
28         frm.inPaciente.focus() // posiciona o cursor no campo
29         return // retorna ao form
30     }
31
32     const nome = frm.inPaciente.value // obtém nome do paciente
33     pacientes.unshift(nome)          // adiciona paciente no início do vetor
34     let lista = ""                   // string para concatenar pacientes
35
36     // forEach aplicado sobre o array pacientes
37     pacientes.forEach((paciente, i) => (lista += `${i + 1}. ${paciente}\n`))
38     respLista.innerText = lista      // exibe a lista de pacientes na página
39     frm.inPaciente.value = ""        // limpa conteúdo do campo de formulário
40     frm.inPaciente.focus()           // posiciona o cursor no campo
41 })
```

Figura 6 – JS do Exemplo (02) – Parte 2

- *For..of e forEach()*
 - *Exemplo (02): - Parte 3*

```
43 frm.btAtender.addEventListener("click", () => {  
44     // se o tamanho do vetor = 0  
45     if (pacientes.length == 0) {  
46         alert("Não há pacientes na lista de espera")  
47         frm.inPaciente.focus()  
48         return  
49     }  
50  
51     const atender = pacientes.shift() // remove do início da fila (e obtém nome)  
52     respNome.innerText = atender      // exibe o nome do paciente em atendimento  
53     let lista = ""                   // string para concatenar pacientes  
54     pacientes.forEach((paciente, i) => (lista += `${i + 1}. ${paciente}\n`))  
55     respLista.innerText = lista      // exibe a lista de pacientes na página  
56 })
```

Figura 7 – JS do Exemplo (02) – Parte 3

- *For..of e forEach()*

- *Observações:*

- *No início do programa JS, criamos uma referência aos elementos a serem manipulados e declaramos um vetor de escopo global, que será utilizado pelas três funções do programa*
 - *Na função associada ao botão **submit** do form, é necessário obter o nome do paciente e utilizar o método **push()** – que insere o nome no final do vetor **pacientes***
 - *Na sequência, o comando **for** percorre todos os itens do vetor, exibindo a lista os pacientes na fila de espera*
 - *Observe que, dentro do laço **for**, o valor da variável **i** é adicionado a **1**, pois ficaria estranho numerar a lista considerando a posição real dos elementos do vetor (**início 0**)*

- *For..of e forEach()*

- *Observações:*

- A *programação associada ao clique no botão Urgência* inicia por uma condição: `if(!form.checkValidity())`, ou então `if (frm.checkValidity() == false)`
 - As *regras de validação adicionadas aos campos do form*, como *required*, *min*, *max*, são *avaliadas apenas quando o botão submit é clicado*
 - Quando utilizamos um *button* e quisermos *validar o form*, devemos *acrescentar essa condição*
 - Após o teste condicional, deve-se *inserir o nome do paciente no início da fila* (o *método unshift(nome)* é utilizado)

- *For..of e forEach()*

- *Observações:*

- *Para acrescentar a lista dos pacientes em **espera**, no botão **Urgência** e **Atender**, recorreremos ao **método** **forEach()** – que é uma forma mais “elegante” de percorrer um vetor quando necessitamos obter o conteúdo e o índice do vetor*
 - *Quando necessitamos apenas acessar o conteúdo do vetor, o laço **for..of** é uma boa alternativa*
 - *Observe que podemos definir um ouvinte de evento para os botões que pertencem ao form a partir do **id** de cada um deles, de forma semelhante ao que é feito para referenciar um campo do form*

- *For..of e forEach()*

- *Observações:*

- *No evento de clique no botão **Atender**, observe que ela começa pela verificação do **número** de **elementos** do **vetor**, pois, se o **vetor** estiver **vazio**, não há atendimentos para realizar*
 - *Na sequência, executa-se o **método** **shift()** que retira o **primeiro paciente** da **lista de espera** e o **armazena** na **variável** **atender**, exibida posteriormente*
 - *No final, novamente é necessário apresentar a **lista dos pacientes**, agora sem o **paciente removido***

- **Localizar Conteúdo**

- Como o **número** de **elementos** de um **vetor** pode ser grande, as linguagens de programação dispõem de alguns **métodos** para nos auxiliar no controle de seu conteúdo
- Um desses controles refere-se à verificação da existência ou não de um conteúdo do vetor
 - **indexOf()**: a busca ocorre a partir do início
 - **lastIndexOf()**: a busca é do final até o seu início. Caso o conteúdo exista no vetor, o **número** do **índice** da **primeira ocorrência** desse conteúdo é **retornado**. Caso o conteúdo pesquisado **não** exista no vetor, o valor **-1** é devolvido
 - **includes()**: retorna **verdadeiro** ou **falso**, de acordo com a existência ou não do conteúdo no vetor

- *Localizar Conteúdo*

- *Exemplo:*

```
<script>  
  const idades = [5, 6, 8, 3, 6, 9]  
  console.log(idades.indexOf(6))           // retorna 1  
  console.log(idades.lastIndexOf(6))      // retorna 4  
  console.log(idades.indexOf(7))          // retorna -1  
  console.log(idades.includes(3))         // retorna true  
</script>
```



- **Localizar Conteúdo**

- **Exemplo (03):**

- *O programa utiliza a função matemática `Math.Random()` para gerar um **número aleatório** entre **1** e **100** que deve ser descoberto pelo usuário*
 - *Para evitar que o jogador aposte um número 2x (e **perca uma chance**), faz-se uso do método `includes()`*
 - *O código HTML desse jogo exibe um campo de entrada e três linhas para a exibição de mensagens: o número de erros e o vetor erros, o número de chances ainda disponíveis para o jogador e as dicas para auxiliar na descoberta do número*

- **Localizar Conteúdo**

- **Exemplo (03):**

- *Programa descubra o número, gera um **número aleatório** a cada novo jogo*



Figura 8 – Renderização do HTML do Exemplo (03)

- *Localizar Conteúdo*
 - *Exemplo (03):*

```
1  <!DOCTYPE html>
2  <html lang="pt-br">
3  <head>
4      <meta charset="UTF-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      <link rel="stylesheet" href="../css/estilos.css">
8      <title>Exemplo 02 | bkBank Academy</title>
9  </head>
10 <body>
11     
12     <h1> Jogo Descubra o Número </h1>
13     <form>
14         <p> Número:
15             <input type="number" min="1" max="100" id="inNumero" required autofocus>
16             <input type="submit" value="Apostar" id="btSubmit">
17             <input type="button" value="Jogar Novamente" id="btNovo" class="oculta">
18         </p>
19     </form>
20     <h3> Erros: <span id="outErros"></span></h3>
21     <h3> Chances: <span id="outChances">6</span></h3>
22     <h3 id="outDica" class="italico">Dica: É um número entre 1 e 100</h3>
23     <pre></pre>
24     <script src="../js/exemplo_2.js"></script>
25 </body>
26 </html>
```

Figura 9 – HTML do Exemplo (03)

- *Localizar Conteúdo*
 - *Exemplo (03): - Parte 1*

```
1  const frm = document.querySelector("form")          // obtém elementos da página
2  const respErros = document.querySelector("#outErros")
3  const respChances = document.querySelector("#outChances")
4  const respDica = document.querySelector("#outDica")
5
6  const erros = []                                     // vetor de escopo global com números já apostados
7  const sorteado = Math.floor(Math.random() * 100) + 1 // num aleatório entre 1 e 100
8  const CHANCES = 6                                   // constante com o número máximo de chances
9
```

Figura 10 – JS do Exemplo (03) – Parte 1

- *Localizar Conteúdo*
 - *Exemplo (03): - Parte 2*

```
10 frm.addEventListener("submit", (e) => { // "escuta" evento submit do form
11   e.preventDefault() // evita envio do form
12   const numero = Number(frm.inNumero.value) // obtém número digitado
13
14   if (numero == sorteado) { // se acertou
15     respDica.innerText = `Parabéns!! Número sorteado: ${sorteado}`
16     frm.btSubmit.disabled = true // troca status dos botões
17     frm.btNovo.className = "exibe"
18   } else {
19     if (erros.includes(numero)) { // se número existe no vetor erros (já apostou)
20       alert(`Você já apostou o número ${numero}. Tente outro...`)
21     } else {
22       erros.push(numero) // adiciona número ao vetor
23       const numErros = erros.length // obtém tamanho do vetor
24       const numChances = CHANCES - numErros // calcula nº de chances
```

Figura 11 – JS do Exemplo (03) – Parte 2

- *Localizar Conteúdo*
 - *Exemplo (03): - Parte 3*

```
26 // exibe nº de erros, conteúdo do vetor e nº de chances
27 respErros.innerHTML = `${numErros} (${erros.join(", ")}`
28 respChances.innerHTML = numChances
29 if (numChances == 0) {
30     alert("Suas chances acabaram...")
31     frm.btSubmit.disabled = true
32     frm.btNovo.className = "exibe"
33     respDica.innerHTML = `Game Over!! Número Sorteado: ${sorteado}`
34 } else {
35     // usa operador ternário para mensagem da dica
36     const dica = numero < sorteado ? "maior" : "menor"
37     respDica.innerHTML = `Dica: Tente um número ${dica} que ${numero}`
38 }
39 }
40 }
41 frm.inNumero.value = "" // limpa campo de entrada
42 frm.inNumero.focus()   // posiciona cursor neste campo
43 })
44
45 frm.btNovo.addEventListener("click", () => {
46     location.reload() // recarrega a página
47 })
```

Figura 12 – JS do Exemplo (03) – Parte 3

- **Localizar Conteúdo**

- **Observações:**

- São declarados um **vetor** para armazenar os **números apostados**, uma **variável** contendo o **número aleatório** a ser descoberto e uma **constante** com o **número** de **chances** do jogador
 - Por que os declarar com escopo global e não dentro da função (local)?
 - Porque as **variáveis** de **escopo global** permanecem na memória, enquanto a página está carregada, e as **variáveis locais**, somente enquanto a função está em **execução**
 - Dessa forma, se a **variável sorteado** fosse declarada dentro da função, um novo sorteio ocorreria a cada aposta de número feita pelo jogador

- **Localizar Conteúdo**

- **Observações:**

- Evento **submit** do **form** são realizados vários testes. Verifica se o jogador acertou o número
 - Caso **ok**, uma mensagem de parabéns é exibida e os botões trocam de status
 - Senão, deve-se utilizar o método **includes()** para verificar se o número apostado já consta no vetor **erros**
 - Caso verdadeiro, a mensagem de alerta é exibida

- **Localizar Conteúdo**

- **Observações:**

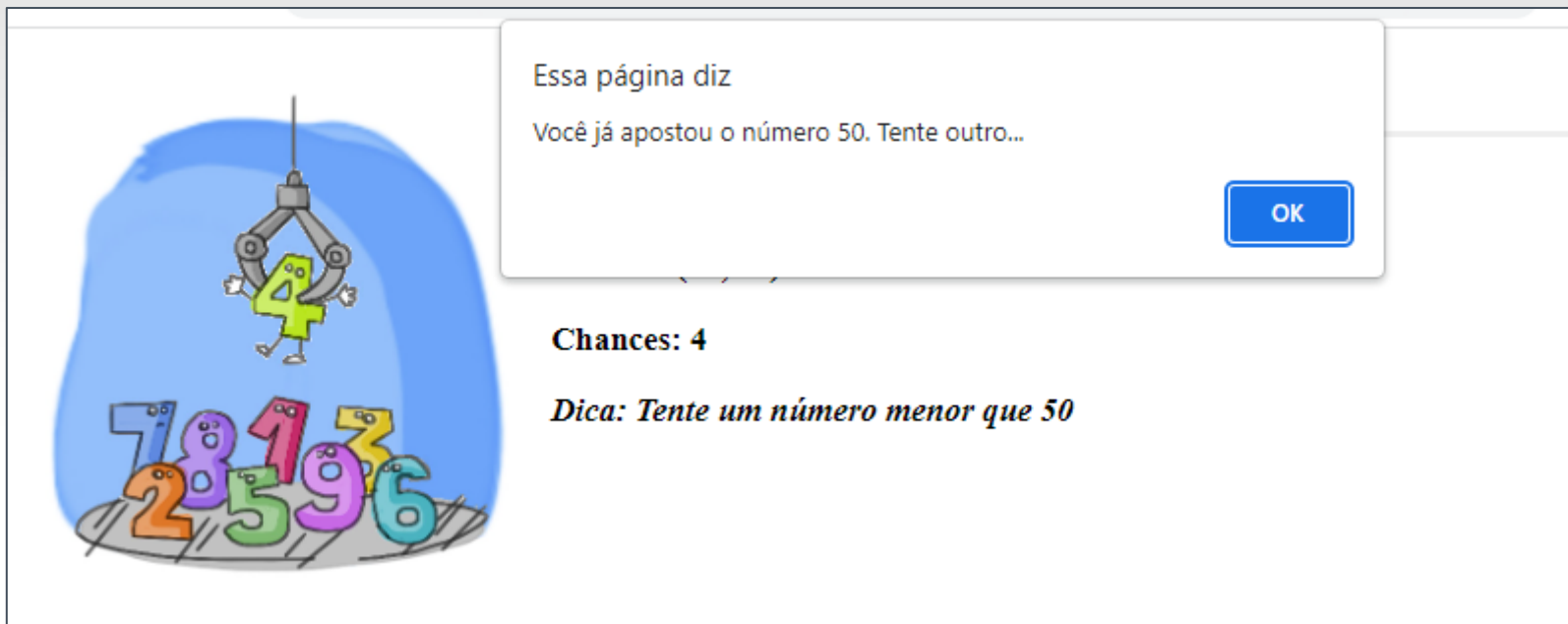


Figura 13 – O método `includes()` verifica se um número já consta no vetor erros

- **Localizar Conteúdo**

- **Observações:**

- As **ações** seguintes **servem** para **adicionar** o **número** ao **vetor** **erros**, **exibir** as **mensagens** na **página** e **verificar** o **número** de **chances** que o **jogador** **ainda** **possui**
 - As **trocas** de **status** dos **botões** **impedem** que o **jogador** **continue** **apostando** outros **números**, mesmo com as **chances** **zeradas**
 - **Apenas** o **botão** “**Jogar Novamente**” **permanece** **disponível** para o **usuário**

- **Localizar Conteúdo**
 - **Observações:**



Figura 14 – Troca de status dos botões impedem novas apostas

- **Localizar Conteúdo**

- **Observações:**

- *O click no botão **Jogar Novamente** contém uma chamada ao método `location.reload()` que recarrega a página*
 - *Dessa forma, um novo número é sorteado, o vetor é zerado e a dica inicial é exibida*

```
frm.btNovo.addEventListener("click", () => {  
    location.reload() // recarrega a página  
})
```

- ***Vetores de Objetos***

- *Um vetor pode conter uma lista de **nomes**, ou de **números***
- *Também é possível definir um vetor que contenha uma **lista** de **objetos**, com alguns **atributos** desse **objeto***
- ***Exemplos:***
 - *objeto **produto**, com os atributos **nome**, **marca** e **preço***
 - *objeto **filme**, com os atributos **título**, **gênero** e **duração***
- *Definir um vetor de objetos nos permite realizar operações sobre esse vetor, como **classificar** os seus elementos por um dos seus atributos*

- ***Vetores de Objetos***

- Um **vetor** de **objetos** é declarado da mesma forma que um **vetor simples**
- Na **inserção** dos **itens** no **vetor**, contudo, devem-se **indicar** os **atributos** que o **compõem**

```
<script>
```

```
  const carros = [ ]
```

```
  carros.push({modelo: "Sander", preco: 46500})
```

```
  carros.push({modelo: "Palio", preco: 37800})
```

```
  for (const carro of carros) {
```

```
    console.log(`${carro.modelo} - R$: ${carro.preco}`)
```

```
  }
```

```
</script>
```

- ***Vetores de Objetos***

- ***Um detalhe na atribuição de dados de um objeto em JS é que, se o nome da variável for **igual** ao do atributo, pode-se omitir a atribuição***

```
const carros = [ ]  
const modelo = "Fiesta"  
const preco = 46800
```

```
carros.push({modelo, preco}) // ou carros.push({modelo: modelo, preco: preco})
```

- ***Desestruturação e Operador Rest/Spread***

- *Um dos acréscimos das novas versões do JS é a possibilidade de atribuir valores às variáveis via desestruturação dos elementos de vetores ou objetos*
- *Utilizaremos o **vetor** de **objetos carros**, com os atributos **modelo** e **preço***
- *Na rotina de apresentação dos dados, pode-se **desestruturar** o objeto*

```
for(const carro of carros) {  
    const {modelo, preço} = carro    // "desestrutura" objeto carro em modelo e  
                                     // preço  
    console.log(`${modelo} - R$: ${preço}`)  
}
```

- **Desestruturação e Operador Rest/Spread**
 - Os atributos **modelo** e **preço** do objeto **carro**, são obtidos a partir de uma única atribuição
 - Como refere a um objeto, os nomes dos atributos devem estar delimitados por **{ }**
 - Para a obtenção de dados de um **Web Service**, em que um objeto contém vários atributos, é uma boa prática explorar essa técnica



- ***Desestruturação e Operador Rest/Spread***
 - ***A desestruturação também pode ocorrer para obter os elementos de um array***

```
const pacientes = ["Ana", "Carlos", "Sofia"]  
const [a, b, c] = pacientes
```

```
console.log(a) // Ana  
console.log(b) // Carlos  
console.log(c) // Sofia
```

- ***Desestruturação e Operador Rest/Spread***

- *Também se pode **desestruturar** os elementos de um vetor com uma **parte** dele sendo **atribuída** a **variáveis** e outra **parte** a um **outro vetor***
- *Para isso, deve-se **utilizar** o operador **Rest(...)** que **cria** um **novo vetor** com os **elementos “restantes”***

```
const pacientes = ["Ana", "Carlos", "João", "Sofia"]  
const [atender, proximo, ...outros] = pacientes
```

```
console.log(atender)      // Ana  
console.log(proximo)     // Carlos  
console.log(outros)      // ["João", "Sofia"]
```


- ***Desestruturação e Operador Rest/Spread***

```
const pacientes = ["Ana", "Carlos", "João", "Sofia"]  
const [atender, proximo, ...outros] = pacientes
```

```
console.log(atender)      // Ana  
console.log(proximo)      // Carlos  
console.log(outros)       // ["João", "Sofia"]
```

- ***Caso o array **pacientes** contenha apenas um nome, ele é atribuído à variável **atender**, **proximo** fica como undefined e o array **outros** fica vazio***
- ***Caso **pacientes** esteja vazio, as variáveis ficam undefined e **outros** = []***

- ***Desestruturação e Operador Rest/Spread***

```
const pacientes = ["Ana", "Carlos", "João", "Sofia"]  
const [atender, proximo, ...outros] = pacientes
```

```
console.log(atender)      // Ana  
console.log(proximo)      // Carlos  
console.log(outros)       // ["João", "Sofia"]
```

- Na ***desestruturação*** dos ***elementos*** de um array, o operador ***Rest*** deve ser o ***último*** da lista de variáveis, justamente pelo fato de ele receber todos os demais elementos ***não*** referenciados pelas variáveis

- ***Desestruturação e Operador Rest/Spread***
 - Os “...” também podem ser utilizados com a ideia de “**espalhar**” os elementos de um **array** ou **objeto** (denominação de **operador Spread**)

```
const carro = { modelo: "Corsa", preco: 59500 }  
const carro2 = { ...carro, ano: 2020 }  
console.log(carro2) // {modelo: "Corsa", preco: 59500, ano: 2020}
```

- ***Desestruturação e Operador Rest/Spread***
 - ***Em aplicações sobre vetores, oferecendo uma forma alternativa para realizar inclusões de elementos no array***

```
let pacientes = ["João", "Sofia"]
```

```
pacientes = ["Anna", ...pacientes] // acrescenta "Ana" no início do vetor
```

```
pacientes = [...pacientes, "Maria"] // acrescenta "Maria" no final
```

- **Desestruturação e Operador Rest/Spread**

- Os “...” podem servir para “**espalhar**” os **elementos** de um **array** ou **objeto** (**Spread**), ou então “**juntar**” **elementos** criando um novo array (**Rest**)
- O operador **Spread** também pode ser **utilizado** para **criar** uma **cópia** com os **elementos** de um **vetor** e, dessa forma, tem um **comportamento** semelhante ao método **slice()** – **sem parâmetros**

```
const pacientes2 = [...pacientes] // ou const pacientes2 = pacientes.slice()
```

- **Pesquisar e Filtrar Dados**
 - Depois de **possuirmos** um **conjunto** de **dados** armazenados em uma **lista**, podemos **exercitar algumas operações frequentemente realizadas** sobre as **listas**, como **pesquisa** ou **filtro** dos **dados**
 - **Exemplos de filtros em um conjunto de dados:**
 - a obtenção do nome e da nota dos alunos aprovados em uma prova
 - clientes com saldo negativo em uma agência bancária
 - contas em atraso de uma empresa
 - Nos **programas** que **realizam** essas **operações** um **cuidado extra** é **necessário**: **informar** ao **usuário** quando uma **pesquisa não** encontrou dados

- *Pesquisar e Filtrar Dados*
 - *Apresentar as **idades** que possuem valor maior ou igual a **18** armazenadas no vetor*

```
<script>
  const idades = [12, 20, 15, 17, 14]
  for(const idade of idades) {
    if (idade >= 18) {
      console.log(idade)
    }
  }
</script>
```

- ***Pesquisar e Filtrar Dados***

- *Como fazer esse script apresentar uma mensagem indicando que não há idades maiores que 18 na lista?*
- *A solução para esse cenário é utilizar uma **variável** de **controle***
- *Essa variável (**flag** ou **sinalizador**), recebe um **valor inicial** antes da repetição*
- *Caso a condição dentro do laço seja **verdadeira**, modifica-se o valor da variável*
- *Após o laço, deve-se verificar, então, se a **variável** mantém o valor inicial*

- *Pesquisar e Filtrar Dados*

- *Isso significa que a condição testada no laço **não** ocorreu e que, portanto, a mensagem indicativa deve ser exibida*

```
<script>
  const idades = [12, 16, 15, 17, 14]
  let maiores = false
  for(const idade of idades) {
    if (idade >= 18) {
      console.log(idade)
      maiores = true
    }
  }
  if(!maiores) {
    console.log("Não há idades maiores que 18 na lista")
  }
</script>
```

- ***Map, Filter e Reduce***

- *São métodos que permitem que operações sobre vetores sejam realizadas de um modo mais eficiente*

- ***Método map()***

```
<script>
```

```
const numeros = [10, 13, 20, 8, 15]    // vetor inicial
```

```
// cada número é obtido e multiplicado por 2, criando um novo array
```

```
const dobros = numeros.map(num => num * 2)
```

```
console.log(dobros.join(", "))        // 20, 26, 40, 16, 30
```

```
</script>
```

- ***Map, Filter e Reduce***

- *Na mesma ideia do **for..of** ou do **forEach()** – no sentido de percorrer cada elemento do vetor, o método **map()** cria um novo vetor com o resultado do processamento realizado sobre cada um dos elementos do vetor original*



- **Map, Filter e Reduce**

- As **operações** podem ser realizadas também sobre **arrays** de **objetos**



The screenshot shows the JS.do Online JavaScript Editor interface. At the top, it says "JS.do Online JavaScript Editor" with the tagline "Edit your code online. Simple, light and fast!". Below this, there are input fields for "Code address:" (with a URL "https://js.do/code/") and "Description:". A toolbar contains buttons for "Run code", "Save", "Add framework" (with a dropdown arrow), and "Create new code". The main code editor area shows the following JavaScript code:

```
1 <script>
2   const amigos = [{nome: "Ana", idade: 20},
3                   {nome: "Bruno", idade: 17},
4                   {nome: "Cátia", idade: 25}]
5
6   const amigos2 = amigos.map(aux => ({nome: aux.nome, nasc: 2022 - aux.idade}))
7
8   for (const amigo of amigos2){
9     console.log(`${amigo.nome} - Nasceu em ${amigo.nasc}`)
10  }
11 </script>
```

Figura 15 – Uso do map() sobre um array de objetos

- **Map, Filter e Reduce**

- As **operações** podem ser **realizadas** também sobre **arrays** de **objetos**

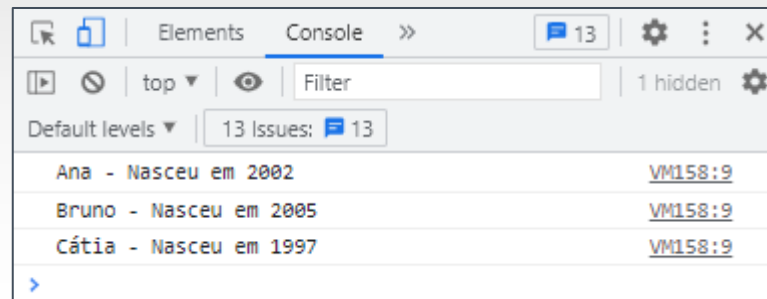


Figura 16 – Resultado do uso do map() sobre um array de objetos

- *Map, Filter e Reduce*

- ***filter()** também cria um novo array, onde cada elemento do vetor de origem é submetido a uma condição, que, se verdadeira, adiciona um elemento ao novo array*

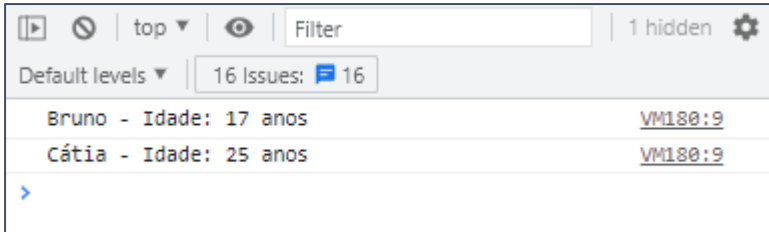
```
<script>
  const numeros = [10, 13, 20, 8, 15]    // vetor inicial
  const pares = numeros.filter(num => num % 2 == 0) // condição para o filtro
  console.log(pares.join(', '))          // 10, 20, 8
</script>
```

- **Map, Filter e Reduce**
 - A *mesma ideia* vale para um **vetor de objetos**
 - O **filtro** é **definido** a partir de **um ou mais atributos** do **objeto**
 - As **condições** podem **conter** os **operadores lógicos** **&&**, **//** ou **!** (**and**, **or** ou **not**), bem como os **demaís métodos**



- *Map, Filter e Reduce*
 - *Aplicando o método **filter()** em um array de objetos*

```
1 <script>
2   const amigos = [{nome: "Ana", idade: 20},
3                   {nome: "Bruno", idade: 17},
4                   {nome: "Cátia", idade: 25}]
5
6   const amigos2 = amigos.filter(aux => aux.idade >= 21 || aux.nome.includes("B"))
7
8   for (const amigo of amigos2){
9     console.log(`${amigo.nome} - Idade: ${amigo.idade} anos`)
10  }
11 </script>
```



Bruno - Idade: 17 anos	VM180:9
Cátia - Idade: 25 anos	VM180:9

Figura 17 – Filter cria um novo array de objetos

- ***Map, Filter e Reduce***

- *Exibir mensagem indicando que o filtro **não** obteve registros que atendam ao critério de pesquisa também se torna mais simples com **filter()***
- *Como um novo array é gerado, basta verificar se ele está vazio*

```
if (amigos2.length == 0) {  
    console.log("Não há amigos com essas condições...")  
}
```

- ***Map, Filter e Reduce***

- ***reduce()*** é útil para ***obter valores cumulativos*** (ou concatenados) a ***partir dos dados de um array***

```
<script>
```

```
  const numeros = [10, 13, 20, 8, 15]
```

```
  const soma = numeros.reduce((acumulador, num) => acumulador + num, 0)
```

```
  console.log(`Soma: ${soma}`) // Soma: 66
```

```
</script>
```

- **Map, Filter e Reduce**

- *Caso o último parâmetro (**zero**) **não** seja fornecido ao **reduce()**, a variável **acumulador** é inicializada com o conteúdo do 1º elemento do vetor, e a repetição inicia a partir do 2º elemento*
- *Sem o parâmetro final, os comandos do **reduce()** equivalem a:*

```
let acumulador = numeros[0]

for(let i = 1; i < numeros.length; i++) {
  acumulador = acumulador + numeros[i]
}
const soma = acumulador
```

- *Map, Filter e Reduce*

- *Obtém a **soma** das **idades** e a **concatenação** dos **nomes** de uma **lista de amigos***

```
1 <script>
2   const amigos = [{nome: "Ana", idade: 20},
3                   {nome: "Bruno", idade: 17},
4                   {nome: "Cátia", idade: 25}]
5
6   const soma = amigos.reduce((acumulador, amigo) => acumulador + amigo.idade, 0)
7   const todos = amigos.reduce((acumulador, amigo) => acumulador + amigo.nome + " ", "")
8
9   console.log(`Soma: ${soma}`)
10  console.log(`Todos: ${todos}`)
11 </script>
```

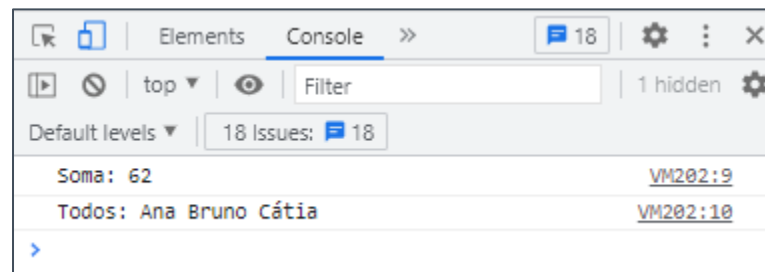


Figura 18 – Método reduce() aplicado sobre um array de objetos

- ***Map, Filter e Reduce***


- *Além de totais, outros valores podem ser obtidos a partir do **método** `reduce()`, já que ele percorre os elementos de um **vetor**, como se **estivéssemos usando um laço for***
- *Identificando o maior valor contido no array*

```
const numeros = [10, 13, 20, 8, 15]
const maior = numeros.reduce((a, b) => Math.max(a, b), 0)
console.log(`Maior: ${maior}`) // Maior: 20
```

- **Map, Filter e Reduce**

- **Exemplo (04):**

- A Revenda bkBank armazena em um **vetor** de **objetos** o **modelo** e o **preço** de carros disponíveis em uma revenda



Revenda bkBank

Modelo:

Preço R\$:

Lista dos Carros Cadastrados

Honda Fit - R\$: 89900.00
Honda Civic - R\$: 113900.00
Ford EcoSport - R\$: 79900.00
Fiat Toro Turbo Diesel - R\$: 125900.00
Volkswagen Amarok - R\$: 223900.00
Volkswagen Polo - R\$: 72900.00
Chevrolet Cruze Turbo - R\$: 95900.00

Figura 19 – Renderização do HTML do Exemplo (04)

- *Map, Filter e Reduce*
 - *Exemplo (04):*

```
1  <!DOCTYPE html>
2  <html lang="pt-br">
3  <head>
4      <meta charset="UTF-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      <link rel="stylesheet" href="../css/estilos.css">
8      <title>Exemplo 03 | bkBank Academy</title>
9  </head>
10 <body>
11     
12     <h1> Revenda bkBank </h1>
13     <form>
14         <p> Modelo: <input type="text" id="inModelo" required autofocus></p>
15         <p> Preço R$:
16             <input type="number" id="inPreco" required>
17             <input type="submit" value="Adicionar">
18         </p>
19         <input type="button" value="Listar Todos" id="btListar">
20         <input type="button" value="Filtrar por Preço" id="btFiltrar">
21         <input type="button" value="Simular Promoção" id="btSimular">
22     </form>
23     <pre></pre>
24     <script src="../js/exemplo_3.js"></script>
25 </body>
26 </html>
```

Figura 20 – HTML do Exemplo (04)

- *Map, Filter e Reduce*
 - *Exemplo (04): - Parte 1*

```
1  const frm = document.querySelector("form")      // obtém elementos da página
2  const resp = document.querySelector("pre")
3
4  const carros = []      // declara vetor global
5
6  frm.addEventListener("submit", (e) => {      // "escuta" evento submit do form
7      e.preventDefault()      // evita envio do form
8      const modelo = frm.inModelo.value      // obtém dados do form
9      const preco = Number(frm.inPreco.value)
10     carros.push({ modelo, preco })      // adiciona dados ao vetor de objetos
11     frm.inModelo.value = ""      // limpa campos do form
12     frm.inPreco.value = ""
13     frm.inModelo.focus()      // posiciona cursor em inModelo
14     // dispara um evento de click em btListar (equivale a um click no botão na página)
15     frm.btListar.dispatchEvent(new Event("click"))
16 })
```

Figura 21 – JS do Exemplo (04) – Parte 1

- *Map, Filter e Reduce*
 - *Exemplo (04): - Parte 2*

```
18 frm.btListar.addEventListener("click", () => { // "escuta" clique em btListar
19     if (carros.length == 0) {                // se tamanho do vetor é igual a 0
20         alert("Não há carros na lista")
21         return
22     }
23
24     // método reduce() concatena uma string, obtendo modelo e preço de cada veículo
25     const lista = carros.reduce((acumulador, carro) =>
26         acumulador + carro.modelo + " - R$: " + carro.preco.toFixed(2) + "\n", "")
27     resp.innerHTML = `Lista dos Carros Cadastrados\n${"".repeat(40)}\n${lista}`
28 })
```

Figura 22 – JS do Exemplo (04) – Parte 2

- *Map, Filter e Reduce*
 - *Exemplo (04): - Parte 3*

```
30 frm.btFiltrar.addEventListener("click", () => {
31     const maximo = Number(prompt("Qual o valor máximo que o cliente deseja pagar?"))
32     if (maximo == 0 || isNaN(maximo)) {           // se não informou ou valor inválido
33         return                                   // ... retorna
34     }
35
36     // cria um novo vetor com os objetos que atendem a condição de filtro
37     const carrosFilter = carros.filter(carro => carro.preco <= maximo)
38
39     if (carrosFilter.length == 0) {               // se tamanho do vetor filtrado é 0
40         alert("Não há carros com preço inferior ou igual ao solicitado")
41         return
42     }
43
44     let lista = ""
45     for (const carro of carrosFilter) {           // percorre cada elemento do array
46         lista += `${carro.modelo} - R$: ${carro.preco.toFixed(2)}\n`
47     }
48     resp.innerText = `Carros Até R$: ${maximo.toFixed(2)}\n${".".repeat(40)}\n${lista}`
49 })
```


Figura 22 – JS do Exemplo (04) – Parte 3

- *Map, Filter e Reduce*
 - *Exemplo (04): - Parte 4*

```
51 frm.btSimular.addEventListener("click", () => {
52     const desconto = Number(prompt("Qual o percentual de desconto: "))
53     if (desconto == 0 || isNaN(desconto)) { // se não informou ou valor inválido
54         return // ... retorna
55     }
56
57     const carrosDesc = carros.map(aux => ({
58         modelo: aux.modelo,
59         preco: aux.preco - (aux.preco * desconto / 100)
60     }))
61
62     let lista = ""
63
64     for (const carro of carrosDesc) { // percorre cada elemento do array
65         lista += `${carro.modelo} - R$: ${carro.preco.toFixed(2)}\n`
66     }
67     resp.innerHTML = `Carros com desconto: ${desconto}%\n${"-"}.repeat(40)}\n${lista}`
68 })
```

Figura 22 – JS do Exemplo (04) – Parte 4

- **Map, Filter e Reduce**
 - **Exemplo (04):**



Essa página diz

Qual o valor máximo que o cliente deseja pagar?

OK Cancelar

Lista dos Carros Cadastrados

Honda Fit - R\$: 89900.00
Honda Civic - R\$: 113900.00
Ford EcoSport - R\$: 79900.00
Fiat Toro Turbo Diesel - R\$: 125900.00
Volkswagen Amarok - R\$: 223900.00
Volkswagen Polo - R\$: 72900.00
Chevrolet Cruze Turbo - R\$: 95900.00

Figura 23 – Solicita o valor limite, para realizar o filtro

- **Map, Filter e Reduce**
 - **Exemplo (04):**



Revenda bkBank

Modelo:

Preço R\$:

Carros Até R\$: 95000.00

Honda Fit - R\$: 89900.00
Ford EcoSport - R\$: 79900.00
Volkswagen Polo - R\$: 72900.00

Figura 24 – Listagem após a aplicação do filtro

- *Classificar os Itens do Vetor*
 - *JS dispõe do **método sort()** para classificar os itens de um vetor em ordem alfabética crescente (o vetor passa a ficar ordenado)*
 - *Para manter a lista na ordem original e apenas apresentar em alguma função do programa os **dados ordenados**, é possível criar uma cópia do vetor original a partir do **método slice()** ou do **operador “...”***
 - *Para classificar os dados em ordem decrescente, podemos utilizar em conjunto os **métodos sort()** e **reverse()***

- *Classificar os Itens do Vetor*
 - ***reverse()***, *inverte a ordem dos elementos de um vetor*

```
<script>  
  const nomes = ["Pedro", "Ana", "João"]  
  
  nomes.sort()  
  console.log(nomes, join(", "))  // Ana, João, Pedro  
  
  nomes.reverse()  
  console.log(nomes, join(", "))  // Pedro, João, Ana  
</script>
```

- ***Classificar os Itens do Vetor***
 - *Em JS os dados do vetor são classificados como **strings**, mesmo que o seu conteúdo seja formado apenas por **números***
 - *Uma classificação de números como strings faz com que o número “**2**” seja considerado maior que “**100**”*
 - *Isso porque a comparação é realizada da esquerda para a direita, caractere por caractere*
 - *Para contornar essa situação, é possível definir uma função para **subtrair** os dados de dois a dois, em cada comparação*

- ***Classificar os Itens do Vetor***
 - ***Torna-se imprescindível entender a sintaxe utilizada para que a ordenação de uma lista de números funcione como o esperado***

```
<script>
  const numeros = [50, 100, 2]

  numeros.sort()
  console.log(numeros, join(", ")) // 100, 2, 50

  numeros.sort((a, b) => a - b)
  console.log(numeros.join(", ")) // 2, 50, 100
</script>
```

- ***Classificar os Itens do Vetor***

- ***Exemplo (05):***

- *O síndico de um determinado condomínio deseja criar uma brinquedoteca no salão do condomínio*
 - *Para tanto, necessita de um programa que leia **nome** e **idade** de crianças e exiba o número e o percentual de crianças em cada idade, a fim de que os brinquedos sejam comprados de acordo com a faixa etária delas*
 - *O programa deve armazenar os dados em um vetor de registros e apresentar o resumo conforme solicitado*

- ***Classificar os Itens do Vetor***
 - ***Exemplo (05):***
 - ***Programa Brinquedoteca do Condomínio***



Programa Brinquedoteca


Nome da Criança:

Idade:

Lucas - 4 anos
Ana - 3 anos
Bianca - 6 anos
Pedro - 6 anos
João - 4 anos
Marina - 4 anos
Juliana - 3 anos
Bruno - 6 anos

Figura 25 – Renderização do HTML do Exemplo (05)

- ***Classificar os Itens do Vetor***
 - ***Exemplo (05):***
 - ***Programa Brinquedoteca do Condomínio***



Programa Brinquedoteca

Nome da Criança:

Idade:

3 ano(s): 2 criança(s) - 25.00%
(Ana, Juliana)

4 ano(s): 3 criança(s) - 37.50%
(Lucas, João, Marina)

6 ano(s): 3 criança(s) - 37.50%
(Bianca, Pedro, Bruno)

Figura 26 – As crianças são agrupadas pela idade

- *Classificar os Itens do Vetor*
 - *Exemplo (05):*

```
1  <!DOCTYPE html>
2  <html lang="pt-br">
3  <head>
4      <meta charset="UTF-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      <link rel="stylesheet" href="../css/estilos.css">
8      <title>Exemplo 04 | bkBank Academy</title>
9  </head>
10 <body>
11     
12     <h1> Programa Brinquedoteca </h1>
13     <form>
14         <p> Nome da Criança:
15             <input type="text" id="inNome" required autofocus>
16         </p>
17         <p> Idade:
18             <input type="number" id="inIdade" min="0" required>
19             <input type="submit" value="Adicionar">
20         </p>
21         <input type="button" value="Listar Todos" id="btListar">
22         <input type="button" value="Resumor por Idade" id="btResumir">
23     </form>
24     <pre></pre>
25     <script src="../js/exemplo_4.js"></script>
26 </body>
27 </html>
```

Figura 27 – HTML do Exemplo (05)

- *Classificar os Itens do Vetor*
 - *Exemplo (05): - Parte 1*

```
1  const frm = document.querySelector("form") // obtém elementos a serem manipulados
2  const resp = document.querySelector("pre")
3
4  const crianas = [] // declara vetor global
5
6  frm.addEventListener("submit", (e) => {
7    e.preventDefault() // evita envio do form
8    const nome = frm.inNome.value // obtém conteúdo dos campos
9    const idade = Number(frm.inIdade.value)
10   crianas.push({ nome, idade }) // adiciona dados ao vetor de objetos
11   frm.reset() // limpa campos do form
12   frm.inNome.focus() // posiciona no campo de formulário
13   frm.btListar.dispatchEvent(new Event("click")) // dispara click em btListar
14 })
```

Figura 28 – HTML do Exemplo (05) – Parte 1

- *Classificar os Itens do Vetor*
 - *Exemplo (05): - Parte 2*

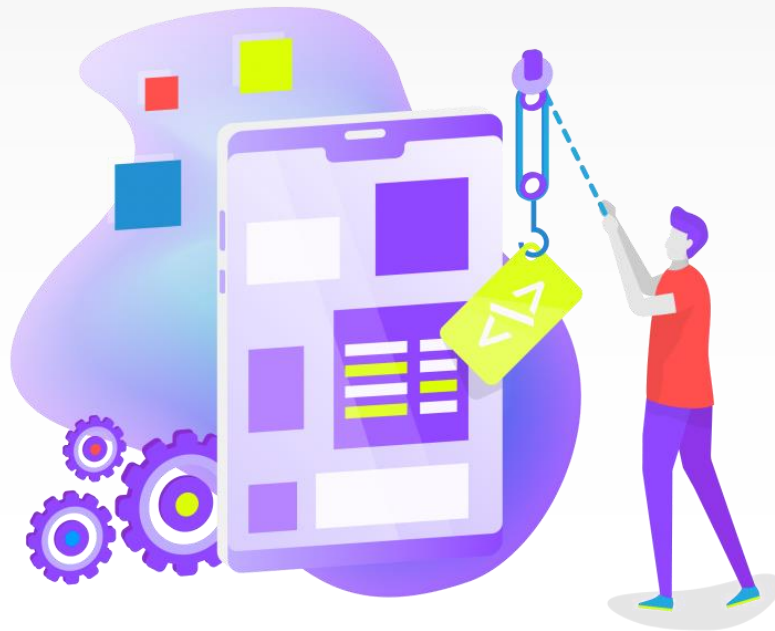
```
16 frm.btListar.addEventListener("click", () => {
17     if (criancas.length == 0) { // se vazio, exibe alerta
18         alert("Não há crianças na lista")
19         return
20     }
21     let lista = "" // para concatenar lista de crianças
22     for (const crianca of criancas) {
23         const { nome, idade } = crianca // desestrutura o objeto
24         lista += nome + " - " + idade + " anos\n"
25     }
26     resp.innerText = lista
27 })
28
29 frm.btResumir.addEventListener("click", () => {
30     if (criancas.length == 0) {
31         alert("Não há crianças na lista")
32         return
33     }
34     const copia = [...criancas] // cria cópia do vetor criança
35     copia.sort((a, b) => a.idade - b.idade) // ordena pela idade
36     let resumo = "" // para concatenar saída
37     let aux = copia[0].idade // menor idade do vetor ordenado
38     let nomes = [] // para inserir nomes de cada idade
39 }
```

Figura 29 – HTML do Exemplo (05) – Parte 2

- *Classificar os Itens do Vetor*
 - *Exemplo (05): - Parte 3*

```
40 for (const crianca of copia) {
41     const { nome, idade } = crianca
42     if (idade == aux) {                                // se é da mesma idade auxiliar...
43         nomes.push(nome)                                // adiciona ao vetor nomes
44     }
45     else {                                              // senão, monta o resumo para cada idade
46         resumo += aux + " ano(s): " + nomes.length + " criança(s) - "
47         resumo += ((nomes.length / copia.length) * 100).toFixed(2) + "%\n"
48         resumo += "(" + nomes.join(", ") + ")\n\n"
49         aux = idade                                    // obtém a nova idade na ordem
50         nomes = []                                    // limpa o vetor dos nomes
51         nomes.push(nome)                                // adiciona o primeiro da nova idade
52     }
53 }
54 // adiciona a última criança
55 resumo += aux + " ano(s): " + nomes.length + " criança(s) - "
56 resumo += ((nomes.length / copia.length) * 100).toFixed(2) + "%\n"
57 resumo += "(" + nomes.join(", ") + ")\n\n"
58 resp.innerHTML = resumo                                // exibe a resposta
59 })
```

Figura 30 – HTML do Exemplo (05) – Parte 3



EXERCÍCIOS

Referências

Duckett, J.; Javascript e JQuery - Desenvolvimento de interfaces web interativas. Alta Books, 2018.

Flanagan, D.; JavaScript: The Definitive Guide, 7th Edition. O'Reilly Media, Inc. 2020.

Scott A. D., MacDonald M., Powers S.; JavaScript Cookbook, 3rd Edition. O'Reilly Media, Inc. 2021.

