



# JavaScript



# Aula 3

# Considerações iniciais

- Diferenças entre passar uma função e executa-la.
- É possível retornar objetos ou arrays em funções. É útil pra quando queremos retornar mais de um valor.
- Variáveis declaradas com let ou const não podem ser usadas antes de serem declaradas.



# Arrays

[referência1 ↗](#)

[referência2 ↗](#)



# Arrays são objetos

- Arrays também são objetos, ou seja, tem [propriedades e métodos intrínsecos](#).



# Encadeamento de métodos

- Se um método retornar um array, é possível utilizar outro método nele.

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

numbers
  .filter((number) => number % 2 == 0)
  .map((evenNumber) => evenNumber + 1)
  .forEach((oddNumber) => console.log(oddNumber))
```

# Arrays multidimensionais

- Também é possível criar arrays multidimensionais (matrizes).

```
const matrix = [  
  ['a11', 'a12', 'a13'], // matrix[0]  
  ['a21', 'a22', 'a23'], // matrix[1]  
  ['a31', 'a32', 'a33'], // matrix[2]  
] // [0] [1] [2]  
  
matrix[1][2] // 'a23'
```

# Arrays são referências

- Arrays também guardam uma referência para seus itens, então seu comportamento em atribuições e passagem de parâmetros é igual ao de objetos.

```
const fruits = ['apple', 'pear']

fruits.length // 2

const addMelon = (fruits) => {
  fruits.push('melon')
}

addMelon(fruits)

fruits.length // 3
```

```
const fruits1 = ['apple', 'pear']

const fruits2 = fruits1

fruits1 === fruits2 // true
```





# Laços e iterações

[referência ↗](#)





- Laços oferecem um jeito fácil e rápido de executar uma ação repetidas vezes. São úteis quando queremos iterar sobre arrays.
- No JavaScript, há diferentes formas de iteração, algumas delas são:
  - for
  - for...in
  - for...of
  - while
  - do...while



# for

- Um laço for é repetido até que a condição especificada seja falsa.
- O laço for no JavaScript é similar ao Java e C.

```
const fruits = ['apple', 'pear', 'melon', 'banana']  
  
for (let index = 0; index < array.length; index++) {  
  const fruit = array[index];  
  console.log(fruit)  
}
```

Leia-se: para index igual a 0, enquanto o index for menor que array.length, incremento o index



# for...in

- A declaração for...in executa iterações a partir de uma variável específica, percorrendo todas as propriedades de um objeto.
- Para cada propriedade distinta, o JavaScript executará uma iteração.

```
for (const key in object) {  
  console.log('key:', key)  
  
  const element = object[key]  
  console.log('element:', element)  
}
```

```
for (const index in array) {  
  console.log('index:', index)  
  
  const item = array[index]  
  console.log('item:', item)  
}
```



# for...of

- A declaração for...of cria uma laço com objetos iterativos (incluindo, Array, Map, Set, assim por conseguinte ), executando uma iteração para o valor de cada propriedade distinta.
- Enquanto o for...in interage com o nome das propriedades, o for...of interage com o valor das propriedades.

```
const fruits = ['apple', 'pear', 'melon', 'banana']

for (const fruit of fruits) {
  console.log('fruit:', fruit)
}
```

# while

- Uma declaração while executa suas instruções, enquanto uma condição especificada seja avaliada como verdadeira.
- O teste da condição ocorre antes que o laço seja executado. Desta forma se a condição for verdadeira o laço executará e testará a condição novamente. Se a condição for falsa o laço termina e passa o controle para as instruções após o laço.

O `while` a seguir executará enquanto `n` for menor que três:

```
n = 0;
x = 0;
while (n < 3) {
    n++;
    x += n;
}
```





# do...while

- Igual o while, mas a verificação da condição ocorre após laço ser executado.

```
do {  
  i += 1;  
  console.log(i);  
} while (i < 5);
```







# break e continue

[break ↗](#)

[continue ↗](#)



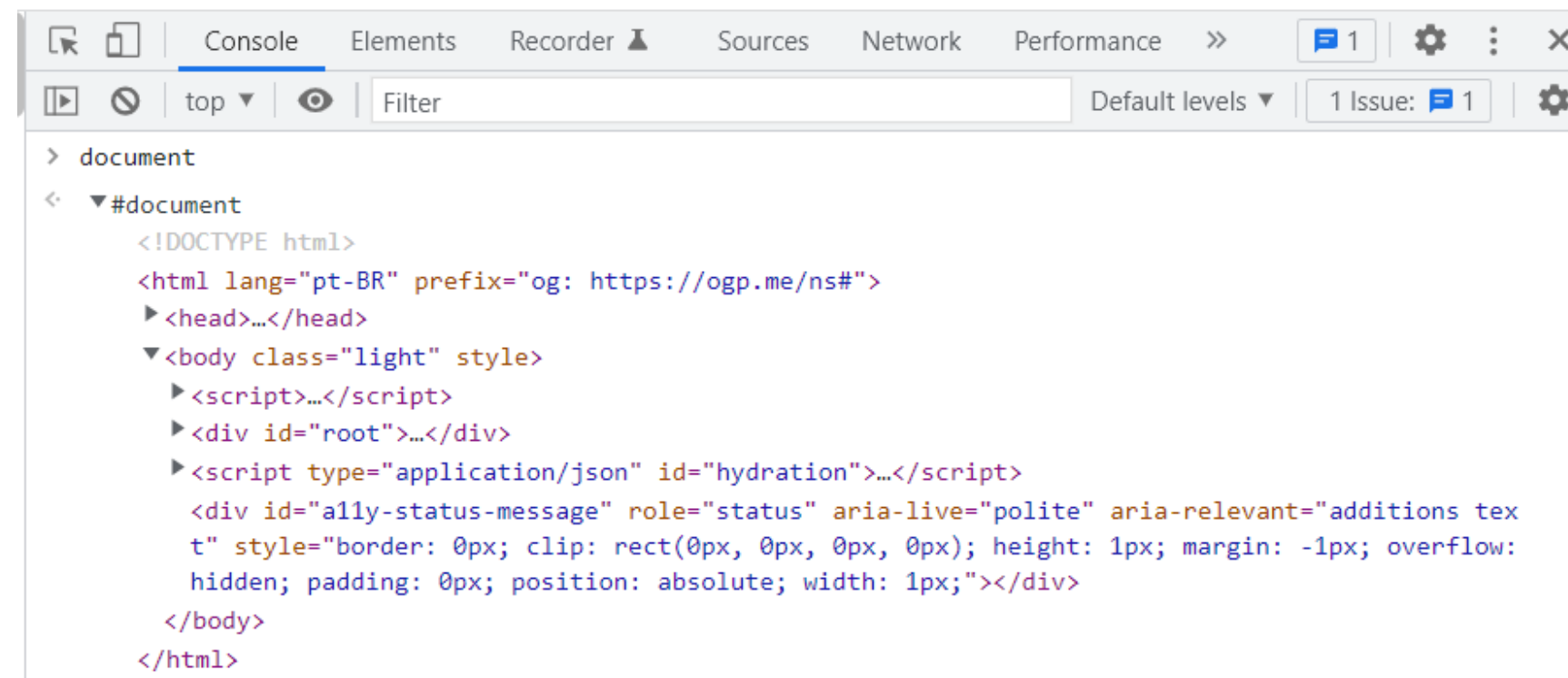
**Dúvidas?**



**0 DOM**



- Para cada página carregada no browser, existe um objeto Document.
- A interface Document serve como um ponto de entrada para o conteúdo da Página ( a árvore DOM, incluindo elementos como <body> e <table>) e provê funcionalidades globais ao documento (como obter a URL da página e criar novos elementos no documento).
- Ele é uma propriedade do window, ou seja, está disponível globalmente.



```
> document
< #document
  <!DOCTYPE html>
  <html lang="pt-BR" prefix="og: https://ogp.me/ns#">
    <head>...</head>
    <body class="light" style>
      <script>...</script>
      <div id="root">...</div>
      <script type="application/json" id="hydration">...</script>
      <div id="ally-status-message" role="status" aria-live="polite" aria-relevant="additions text" style="border: 0px; clip: rect(0px, 0px, 0px, 0px); height: 1px; margin: -1px; overflow: hidden; padding: 0px; position: absolute; width: 1px;"></div>
    </body>
  </html>
```

[referência1 ↗](#)

[referência2 ↗](#)

# Pegando elementos





- A partir do document, podemos trazer elementos do HTML para o JavaScript como objetos.
- Para isso, temos alguns métodos disponíveis:
  - `querySelector()`
  - `querySelectorAll()`
  - `getElementById()`
  - `getElementsByClassName()`
  - `getElementsByTagName()`



# querySelector()

- Retorna o primeiro elemento dentro do documento que corresponde ao grupo especificado de seletores.
- Podemos usar qualquer seletor CSS.

```
const paragraph1 = document.querySelector('p')
const paragraph2 = document.querySelector('p#paragraph')
const paragraph3 = document.querySelector('p#paragraph.text')
const paragraph4 = document.querySelector('p[title="parágrafo"]')

console.log(paragraph4) // <p id="paragraph" class="text" title="parágrafo">Meu parágrafo</p>
```



# querySelectorAll()

- Retorna uma lista de elementos presentes no documento que coincidam com o grupo de seletores especificado. O objeto retornado é uma [NodeList](#).
- Também podemos usar qualquer seletor CSS.

```
<ul>
  <li>item 1</li>
  <li>item 2</li>
  <li>item 3</li>
  <li>item 4</li>
</ul>
```

```
const itemList = document.querySelectorAll('li')

itemList.forEach((itemElement) => {
  console.log(itemElement)
})
```





# getElementById()

- Retorna a referência do elemento através do seu ID. Se não existe um elemento com o id fornecido, esta função retorna null.

```
const paragraph = document.getElementById('paragraph')  
console.log(paragraph) // <p id="paragraph">Meu parágrafo</p>
```

# getElementsByClassName()

- Retorna uma [HTMLCollection](#) de elementos que possuem o nome da classe dada.

```
<ul>
  <li class="item">item 1</li>
  <li class="item">item 2</li>
  <li class="item">item 3</li>
  <li class="item">item 4</li>
  <li class="item">item 5</li>
</ul>
```

```
const itemList = document.getElementsByClassName('item')

for (const itemElement of itemList) {
  console.log(itemElement)
}

// ou

Array.from(itemList).forEach((itemElement) => {
  console.log(itemElement)
})
```



# getElementsByTagName()

- Retorna um HTMLCollection de elementos com o nome da tag fornecido.

```
<ul>
  <li class="item">item 1</li>
  <li class="item">item 2</li>
  <li class="item">item 3</li>
  <li class="item">item 4</li>
  <li class="item">item 5</li>
</ul>
```

```
const itemList = document.getElementsByTagName('li')

for (const itemElement of itemList) {
  console.log(itemElement)
}

// ou

Array.from(itemList).forEach((itemElement) => {
  console.log(itemElement)
})
```



# Encadeamento dos métodos

- Podemos usar esses métodos (exceto o "getElementById") também em elementos, não só no document. Nesse caso, eles fazem a busca por elementos que estejam dentro do elemento atual.

```
<body>
  <p>paragraph <span>1</span></p>
  <div class="container">
    <p id="second-paragraph">paragraph <span>2</span></p>
    <p>paragraph <span>3</span></p>
  </div>
</body>
```

```
const container = document.getElementsByClassName('container')[0]
const secondParagraph = container.querySelector('#second-paragraph')
const secondParagraphSpan = secondParagraph.getElementsByTagName('span')[0]

// ou

const secondParagraphSpan2 = document
  .getElementsByClassName('container')[0]
  .querySelector('#second-paragraph')
  .getElementsByTagName('span')[0]
```



**Dúvidas?**

# Manipulando elementos





- element.classList
  - element.classList.add('class')
  - element.classList.remove('class')
  - element.classList.toggle('class')
- element.className
- element.getAttribute('attribute')
- element.setAttribute('attribute', value)
- element.removeAttribute('attribute')
- element.lastElementChild
- element.firstElementChild
- element.children
- element.innerHTML
- element.textContent
  - A propriedade textContent recebe apenas texto puro e as tags HTML inseridas por ela não são interpretadas.
- element.style
  - element.style.cssText = ``
- element.id
- element.value





# Adicionando e removendo elementos





- `document.createElement()`
- `document.createTextNode()`
- `element.appendChild()`
- `element.insertAdjacentElement()`
- `element.removeChild()`
- `element.remove()`



**Dúvidas?**



# Eventos



- Eventos são ações ou ocorrências que acontecem no sistema que estamos desenvolvendo, no qual este te alerta sobre essas ações para que você possa responder de alguma forma, se desejado. Por exemplo, se o usuário clica em um botão numa página web, você pode querer responder a esta ação mostrando na tela uma caixa de informações. dhdhddhjdhjdhdhddddd
- Ouvindo eventos
  - `element.addEventListener()`
  - `element.removeEventListener()`
- Eventos
  - `click`
  - `change`
  - `submit`
- Objeto event
  - `event.target`
  - `event.target.value`
  - `event.preventDefault()`
  - `event.stopPropagation()`

[referência1 ↗](#)

[referência2 ↗](#)



# Fim!

