



FUNDAMENTOS DA PROGRAMAÇÃO WEB

AULA 6



Prof.^a Margarete Klamas



CONVERSA INICIAL

Vamos avançar um pouco mais em nossos estudos de *JavaScript*. O objetivo desta abordagem é fazer um aprofundamento em *JavaScript*.

Ressaltando a observação feita anteriormente: vamos estabelecer como prática o padrão da web: utilizar HTML para a estruturação de conteúdo. Ao formatar, utilize CSS e o comportamento é feito com *JavaScript*.

No tópico 1, abordaremos tipos de dados: objetos e *arrays*. No tópico 2, estudaremos funções. No tópico 3, veremos como utilizar *BOM*. No tópico 4, veremos como utilizar o DOM. No tópico 5, abordaremos eventos.

Bons Estudos!

TEMA 1 – TIPO DE DADOS

Estudamos, anteriormente, que literais são uma forma de anotar valores (dados) no código do programa. Conhecemos os tipos de dados primitivos: *boolean*, *number*, *BigInt*, *string*, *undefined*. Agora iremos iniciar nossos estudos com os principais tipos de dados literais mais complexos. Iremos falar de **objetos** e ***arrays***.

1.1 Objetos

Objetos são as estruturas de dados conhecidas como registro. Um registro é uma coleção não ordenada de campos nomeados. Cada campo tem seu próprio nome (ou chave) e um valor atribuído. No caso de objetos *JavaScript*, esses campos geralmente são chamados de propriedades. Registros, ou no nosso caso, objetos, permitem armazenar vários valores de diferentes tipos em um só lugar. Em *JavaScript*, existem algumas maneiras de criar objetos, mas a mais fácil e rápida é usar o literal de colchetes `{}`. Exemplos:

```
let exeObj = {  
  nr: 500,  
  str: "palavra"  
};
```

Ou objeto vazio:

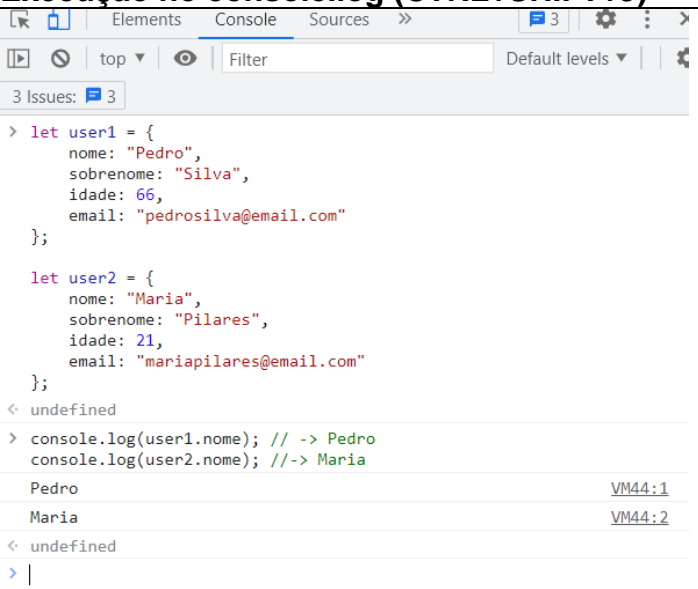
```
let carro={};
```



Observe que criamos objetos usando o mesmo literal, mas ao mesmo tempo criamos propriedades que são pares chave-valor. Podemos acessar esses registros individualmente:

```
console.log(exeObj.nr); // -> 500
console.log(exeObj.str); // -> palavra
```

Exemplo de uso de objetos:

Exemplo	Execução no console.log (CTRL+SHIFT+J)
<pre>let user1 = { nome: "Pedro", sobrenome: "Silva", idade: 66, email: "pedrosilva@email.com" }; let user2 = { nome: "Maria", sobrenome: "Pilares", idade: 21, email: "mariapilares@email.com" };</pre>	
<p>Podemos acessar os dados individualmente:</p> <pre>console.log(user1.nome); // -> Pedro console.log(user2.nome); //-> Maria</pre>	

1.1.1 Como criar um objeto

Para criar um objeto, vamos utilizar o operador unitário **new**. Este operador criar uma instância do objeto nativo ou do objeto definido.

Sintaxe:

nomeObjeto = new construtor(argumento);

Exemplos:

```
data=new Data();
let livro = new Object();
```

O construtor *Object()* é nativo da linguagem e cria um objeto genérico vazio.



Dentre as propriedades que um livro possui, podemos citar: título, autor, editora, ano de publicação, edição, páginas, preço, frete (a calcular), capítulos. Vamos criar esse objeto. Sintaxe formal:

Exemplo	Execução no console.log (CTRL+SHIFT+J)
<pre>var livro = new Object(); livro.titulo="A Bela e a Adormecida"; livro.autor="Neil Gaiman"; livro.editora="Rocco Jovens Leitores"; livro.anoPublicacao=2015; livro.edicao="1ª"; livro.paginas=72; livro.preco="R\$ 30,00"; livro.frete= function(ceporigem, cepdestino,peso){ var valorFrete=" "; //script do calculo frete return valorFrete; } livro.capitulo1="Era o reino mais próximo"; livro.capitulo2="A rainha acordou cedo"; livro.capitulo3="Os três anões emergiram"; livro.capitulo4="-Dormindo? - perguntou a rainha"; livro.capitulo5="Ela cavalgou um dia inteiro"; livro.capitulo6="O castelo na Floresta de Acaire";</pre>	<pre>> var livro = new Object(); livro.titulo="A Bela e a Adormecida"; livro.autor="Neil Gaiman"; livro.editora="Rocco Jovens Leitores"; livro.anoPublicacao=2015; livro.edicao="1ª"; livro.paginas=72; livro.preco="R\$ 30,00"; livro.frete= function(ceporigem, cepdestino,peso){ var valorFrete=" "; //script do calculo frete return valorFrete; } livro.capitulo1="Era o reino mais próximo"; livro.capitulo2="A rainha acordou cedo"; livro.capitulo3="Os três anões emergiram"; livro.capitulo4="-Dormindo? - perguntou a rainha"; livro.capitulo5="Ela cavalgou um dia inteiro"; livro.capitulo6="O castelo na Floresta de Acaire"; < 'O castelo na Floresta de Acaire'</pre>

O objeto possui treze propriedades e um método frete, que calcula o valor do frete. A função que define o método para calcular o valor do frete possui três argumentos e retorna o valor do frete. Frete é uma função ilustrativa de como criar um método para um objeto.

Podemos acrescentar novas propriedades ao objeto:

```
livro.novaPropriedade=valor da nova propriedade;
```

A sintaxe para **exibir os valores de propriedades e métodos** de um **objeto** é a mesma para exibir valores de variáveis. Exemplo:

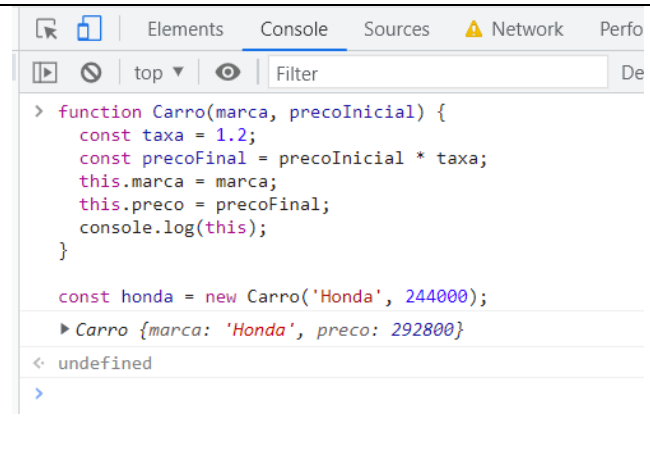


Exemplo	Execução no console.log (CTRL+SHIFT+J)
<pre>var nomeAutor=livro.autor; var nomeEditora=livro.editora; alert("Autor: " +nomeAutor + "\nEditora: " +nomeEditora) Ou: var nomeAutor1=livro["autor"]; var nomeEditora1=livro["editora"]; alert("Autor: " +nomeAutor1 + "\nEditora: " +nomeEditora1)</pre>	<pre>> var nomeAutor=livro.autor; var nomeEditora=livro.editora; alert("Autor: " +nomeAutor + "\nEditora: " +nomeEditora) < undefined > var nomeAutor1=livro["autor"]; var nomeEditora1=livro["editora"]; alert("Autor: " +nomeAutor1 + "\nEditora: " +nomeEditora1) < undefined ></pre> <p>Obs.: Irá abrir janelas de alerta.</p>

Outro exemplo:

Exemplo	Explicação
<pre>const carro = { marca: 'Marca', preco: 0, } const honda = carro; honda.marca = 'Honda'; honda.preco = 244000; const nissan = carro; nissan.marca = 'Nissan'; nissan.preco = 120000;</pre>	<p>Carro, Honda e Nissan apontam para o mesmo objeto. Podemos utilizar funções construtoras para criar objetos.</p>
Funções Construtoras	
<pre>function Carro() { this.marca = 'Marca'; this.preco = 0; } const honda = new Carro(); honda.marca = 'Honda'; honda.preco = 244000; const fiat = new Carro(); fiat.marca = 'Nissan'; fiat.preco = 120000;</pre>	<p>Funções construtoras criam novos objetos sempre que chamamos ela.</p> <p>Usar Pascal Case (inicial em maiúscula) para nomear as funções construtoras.</p>
Função Construtora de Objetos com Parâmetros e Argumentos	
<pre>function Carro(marca, preco) { this.marca = marca; this.preco = preco; } const honda = new Carro('Honda', 244000); const nissan = new Carro('Nissan', 120000);</pre>	<p>O uso da palavra this faz referência ao próprio objeto criado.</p>



<pre>function Carro(marca, precoInicial) { const taxa = 1.2; const precoFinal = precoInicial * taxa; this.marca = marca; this.preco = precoFinal; console.log(this); } const honda = new Carro('Honda', 244000);</pre>	
---	--

1.1.2 Como iterar por um objeto

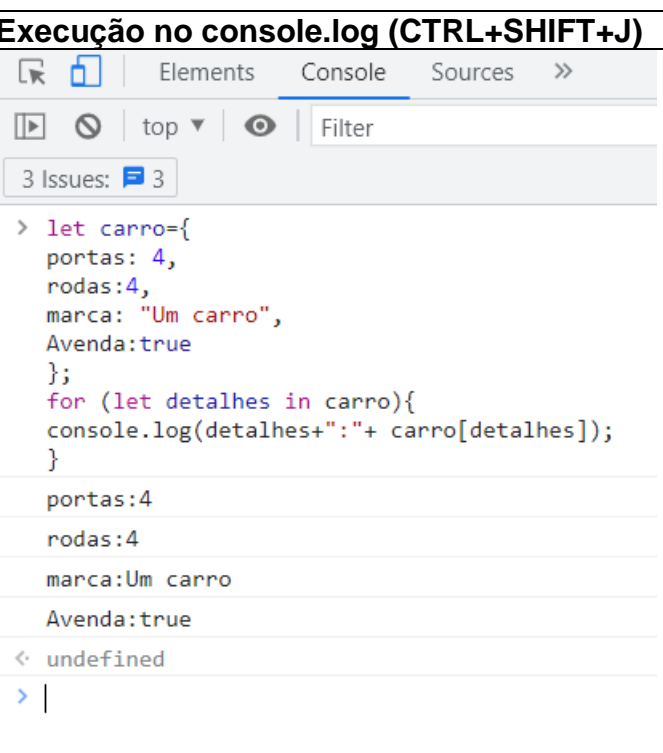
O laço **for...in** interage sobre propriedades enumeradas de um objeto, na ordem original de inserção. O laço pode ser executado para cada propriedade distinta do objeto.

Sintaxe:

```
for (variavel in objeto) {...}
}
```

variável: é um nome de variável

objeto: é o nome do objeto criado.

Exemplo	Execução no console.log (CTRL+SHIFT+J)
<pre>let carro={ portas: 4, rodas:4, marca: "Um carro", Avenda:true }; for (let detalhes in carro){ console.log(detalhes+" "+ carro[detalhes]); }</pre> <p>Obs.: o sinal de + concatena</p>	

No exemplo acima, criamos a **variável detalhes** para exibir a **chave do objeto** e para exibir o **valor** utilizamos o **nome do objeto[variável]**.



1.2 Arrays

Em **arrays**, podemos armazenar os valores, porém não podemos criar uma chave, como fizemos em objetos. Exemplos:

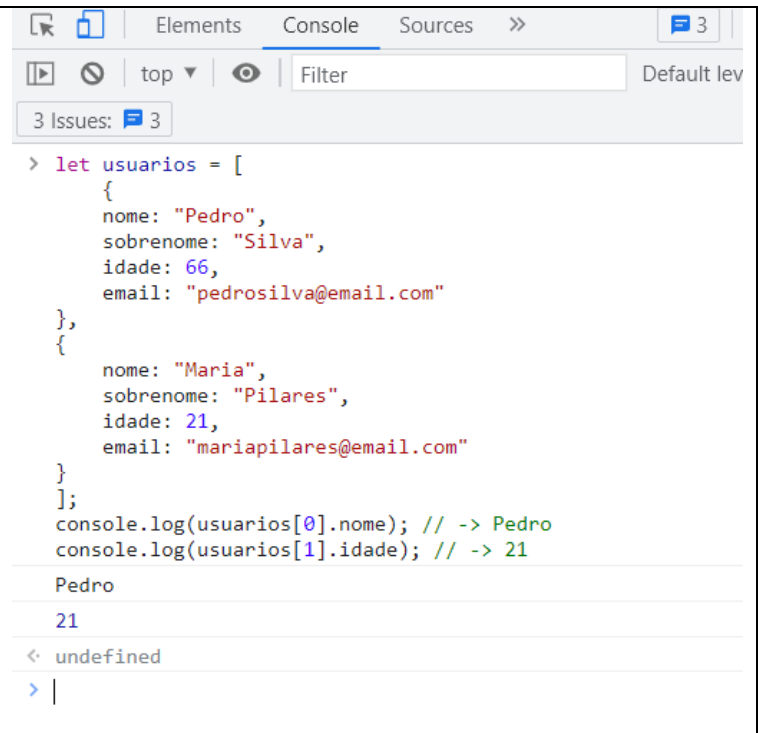
Exemplo	Execução no console.log (CTRL+SHIFT+J)
<pre>let dias = ["Seg", "Ter", "Qua", "Qui", "Sex", "Sab", "Dom"]; console.log(dias[0]); // -> Seg console.log(dias[2]); // -> Qua</pre>	
<p>Podemos criar arrays vazios. Exemplo:</p> <pre>let frutas= []; console.log(frutas[0]); // -> undefined frutas[0] = "banana"; frutas[2] = "maça"; console.log(frutas[0]); // -> banana console.log(frutas[1]); // -> undefined console.log(frutas[2]); // -> maçã</pre>	

Os elementos de um **array** podem ser quaisquer dados, incluindo objetos.

Exemplo de **array**:

Exemplo	Execução no console.log (CTRL+SHIFT+J)
---------	--



<pre>let usuarios = [{ nome: "Pedro", sobrenome: "Silva", idade: 66, email: "pedrosilva@email.com" }, { nome: "Maria", sobrenome: "Pilares", idade: 21, email: "mariapilares@email.com" }]; console.log(usuarios[0].nome); // -> Pedro console.log(usuarios[1].idade); // -> 21</pre>	
---	--

Podemos adicionar um novo usuário ao *array*, atribuindo um novo elemento a um índice:

<pre>usuarios[2] = { nome: "Irene", sobrenome "Souza", idade: 32, email: "irenesouza@email.com" }</pre>	
---	--

Podemos utilizar ***length*** para verificar a quantidade de elementos de um *array*.

```
let nomes = ["Patricia", "Camila", "Mateus", "Samuel"];
console.log(nomes.length); // -> 4
```

1.2.1. Métodos

Podemos utilizar ***indexOf*** para verificar o índice onde se encontra posicionado o valor no *array*:

Exemplo	Execução no console.log (CTRL+SHIFT+J)



```
let nomes = ["Patricia", "Camila", "Mateus", "Samuel"];
console.log(nomes.indexOf("Patricia")); //0
console.log(nomes.indexOf("Samuel")); //3
```

nomes

3	Samuel	← final
2	Mateus	
1	Camila	
0	Patricia	← início

`nomes.indexOf("Samuel") -> 3`

Podemos utilizar **push** para inserir um valor na posição final no *array*. Continuando com o exemplo anterior:

Exemplo	Execução no console.log (CTRL+SHIFT+J)															
<pre>nomes.push("JJ"); console.log(nomes); //['Patricia', 'Camila', 'Mateus', 'Samuel', 'JJ']</pre> <div><p>nomes</p><table><tr><td>4</td><td>JJ</td><td>← final</td></tr><tr><td>3</td><td>Samuel</td><td></td></tr><tr><td>2</td><td>Mateus</td><td></td></tr><tr><td>1</td><td>Camila</td><td></td></tr><tr><td>0</td><td>Patricia</td><td>← início</td></tr></table></div>	4	JJ	← final	3	Samuel		2	Mateus		1	Camila		0	Patricia	← início	
4	JJ	← final														
3	Samuel															
2	Mateus															
1	Camila															
0	Patricia	← início														

O método **unshift** é similar ao método **push** com a diferença que adiciona um elemento na primeira posição do *array*. Exemplo:

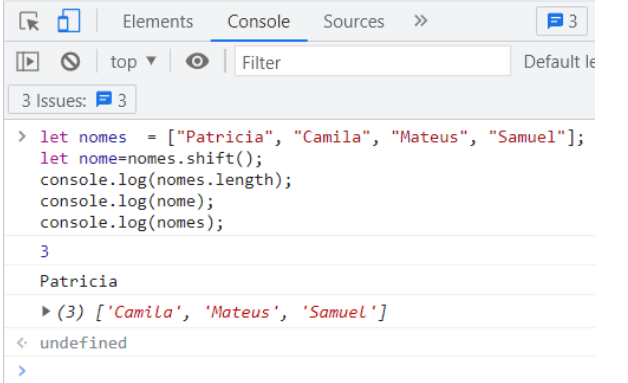
Exemplo	Execução no console.log (CTRL+SHIFT+J)																		
<pre>nomes.unshift("Maria"); ;</pre> <div><p>nomes</p><table><tr><td>5</td><td>JJ</td><td>← final</td></tr><tr><td>4</td><td>Samuel</td><td></td></tr><tr><td>3</td><td>Mateus</td><td></td></tr><tr><td>2</td><td>Camila</td><td></td></tr><tr><td>1</td><td>Patricia</td><td>← início</td></tr><tr><td>0</td><td>Maria</td><td></td></tr></table></div>	5	JJ	← final	4	Samuel		3	Mateus		2	Camila		1	Patricia	← início	0	Maria		
5	JJ	← final																	
4	Samuel																		
3	Mateus																		
2	Camila																		
1	Patricia	← início																	
0	Maria																		

O método **pop()** permite eliminar o último elemento do *array()*. Exemplo:



Exemplo	Execução no console.log (CTRL+SHIFT+J)
<pre>let nomes = ["Patricia", "Camila", "Mateus", "Samuel"]; let nome=nomes.pop(); console.log(nomes.length); // -> 3 console.log(nome); // -> Samuel console.log(nomes); // -> ['Patricia', 'Camila', 'Mateus']</pre>	
<div><div><div>3 Samuel</div><div>2 Mateus</div><div>1 Camila</div><div>0 Patricia</div></div><div><div>2 Mateus</div><div>1 Camila</div><div>0 Patricia</div></div><div>← nomes.pop();</div></div>	

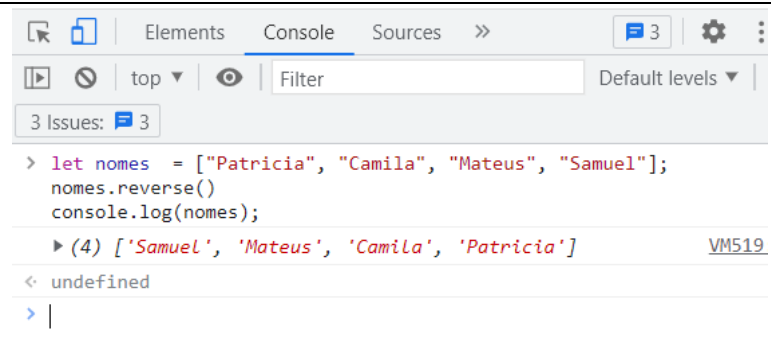

O método **shift()** é similar ao pop, porém permite eliminar o primeiro elemento do array.

Exemplo	Execução no console.log (CTRL+SHIFT+J)
<pre>let nomes = ["Patricia", "Camila", "Mateus", "Samuel"]; let nome=nomes.shift(); console.log(nomes.length); console.log(nome); console.log(nomes);</pre>	
<div><div><div>3 Samuel</div><div>2 Mateus</div><div>1 Camila</div><div>0 Patricia</div></div><div><div>2 Samuel</div><div>1 Mateus</div><div>0 Camila</div></div><div><div>← nomes.shift();</div><div><div>3 Samuel</div><div>2 Mateus</div><div>1 Camila</div><div>0 Patricia</div></div></div></div>	

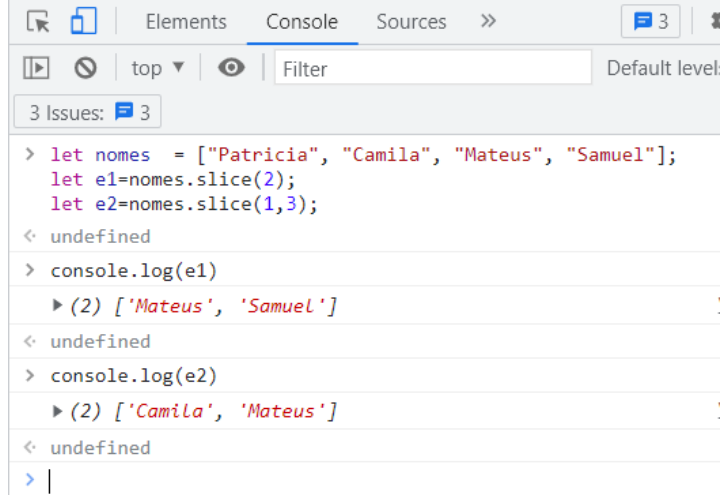
O método **reverse()** inverte a ordem dos elementos do array.

Exemplo	Execução no console.log (CTRL+SHIFT+J)
---------	--

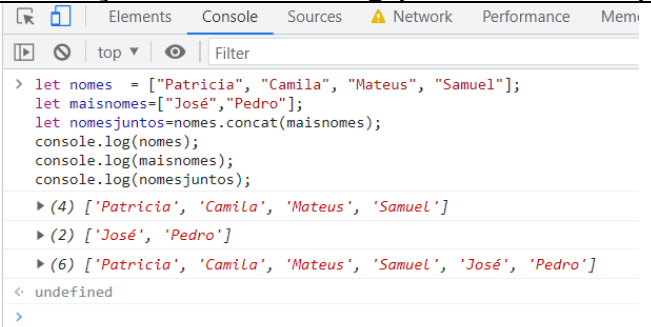


<pre>let nomes = ["Patricia", "Camila", "Mateus", "Samuel"]; nomes.reverse(); console.log(nomes)</pre>																											
<table border="1"><tr><td>3</td><td>Samuel</td></tr><tr><td>2</td><td>Mateus</td></tr><tr><td>1</td><td>Camila</td></tr><tr><td>0</td><td>Patricia</td></tr></table>	3	Samuel	2	Mateus	1	Camila	0	Patricia	<table border="1"><tr><td>3</td><td>Patricia</td></tr><tr><td>2</td><td>Camila</td></tr><tr><td>1</td><td>Mateus</td></tr><tr><td>0</td><td>Samuel</td></tr></table>  <table border="1"><tr><td colspan="2">nomes.reverse();</td></tr><tr><td>3</td><td>Samuel</td></tr><tr><td>2</td><td>Mateus</td></tr><tr><td>1</td><td>Camila</td></tr><tr><td>0</td><td>Patricia</td></tr></table>	3	Patricia	2	Camila	1	Mateus	0	Samuel	nomes.reverse();		3	Samuel	2	Mateus	1	Camila	0	Patricia
3	Samuel																										
2	Mateus																										
1	Camila																										
0	Patricia																										
3	Patricia																										
2	Camila																										
1	Mateus																										
0	Samuel																										
nomes.reverse();																											
3	Samuel																										
2	Mateus																										
1	Camila																										
0	Patricia																										

O método **slice()** permite criar um novo array a partir de elementos selecionados de outro array.

Exemplo	Execução no console.log (CTRL+SHIFT+J)
<pre>let nomes = ["Patricia", "Camila", "Mateus", "Samuel"]; let e1=nomes.slice(2) let e2=nomes.slice(1, 3);</pre>	

O método **concat()** cria um novo array juntando elementos de arrays.

Exemplo	Execução no console.log (CTRL+SHIFT+J)
<pre>let nomes = ["Patricia", "Camila", "Mateus", "Samuel"]; let maisnomes=["José","Pedro"]; let nomesjuntos=nomes.concat(maisnomes); console.log(nomes); console.log(maisnomes); console.log(nomesjuntos);</pre>	



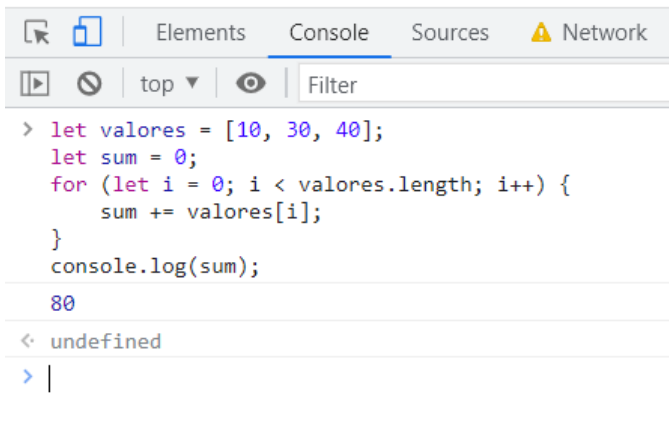
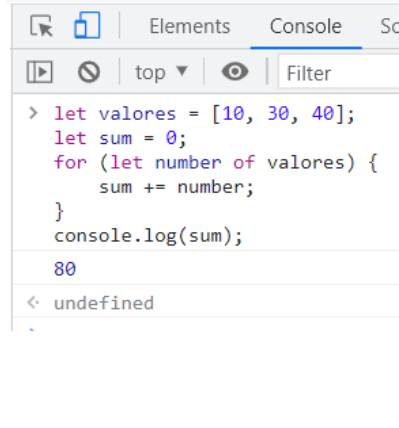
1.2.1. Como iterar por listas (*for ... of*)

Além do *loop* *for* regular, existem duas versões específicas, uma das quais, *for...of*, é dedicada ao uso com *arrays*. Em um *loop* desse tipo, não especificamos explicitamente nenhuma condição ou número de iterações, pois é executado exatamente quantas vezes houver elementos no *array* indicado.

Sintaxe:

```
for (variavel of array) {  
  bloco de código  
}
```

Vamos somar os elementos de um *array*:

Usando <i>for</i>	Usando <i>for ... of</i>
<pre>let valores = [10, 30, 40]; let sum = 0; for (let i = 0; i < valores.length; i++) { sum += valores[i]; } console.log(sum);</pre> 	<pre>let valores = [10, 30, 40]; let sum = 0; for (let number of valores) { sum += number; } console.log(sum);</pre> 
No código acima temos um <i>array</i> de números. Inicializamos a variável <i>sum</i> com valor zero. Utilizamos o <i>for</i> para verificarmos a quantidade de itens do <i>array</i> . Inicializamos a variável <i>i</i> com zero. Estabelecemos que a condição de parada é a quantidade de itens do <i>array</i> (<i>valores.length</i>). Incrementamos mais um item (<i>i++</i>). No bloco de código que executamos, é atribuir à variável <i>sum</i> os valores do <i>array</i> <i>valores</i> . Exibimos no console o resultado.	A diferença do <i>for ... of</i> Criamos a variável <i>number</i> para armazenar cada valor do <i>array</i> <i>valores</i> . Executamos o bloco que soma os valores.



Outro exemplo:

```
let usuarios = [  
  {  
    nome: "Pedro",  
    sobrenome: "Silva",  
    idade: 66,  
    email: "pedrosilva@email.com"  
  },  
  {  
    nome: "Maria",  
    sobrenome: "Pilares",  
    idade: 21,  
    email: "mariapilares@email.com"  
  }  
];  
for (let pessoa of usuarios){  
  console.log(pessoa);  
}
```

```
> let usuarios = [  
  {  
    nome: "Pedro",  
    sobrenome: "Silva",  
    idade: 66,  
    email: "pedrosilva@email.com"  
  },  
  {  
    nome: "Maria",  
    sobrenome: "Pilares",  
    idade: 21,  
    email: "mariapilares@email.com"  
  }  
];  
for (let pessoa of usuarios){  
  console.log(pessoa);  
}
```

```
▶ {nome: 'Pedro', sobrenome: 'Silva', idade: 66, email: 'pedrosilva@email.com'}  
▶ {nome: 'Maria', sobrenome: 'Pilares', idade: 21, email: 'mariapilares@email.com'}  
◀ undefined  
> |
```

No exemplo acima, utilizamos *for....of* para exibir os objetos da lista usuários.

TEMA 2 – FUNÇÕES

Em programação, muitas vezes precisamos executar um determinado trecho de código diversas vezes. Se pensarmos em copiar o trecho quando necessário, estaríamos sendo ineficientes. Além do fato que o tamanho do código iria aumentar.



Uma solução simples para este problema é uma função. Uma **função** é apenas um pedaço separado de código que você nomeia, da mesma forma que você nomeia uma variável. Se você quiser usá-lo em algum lugar, ele é simplesmente referenciado por esse nome (dizemos chamar a função).

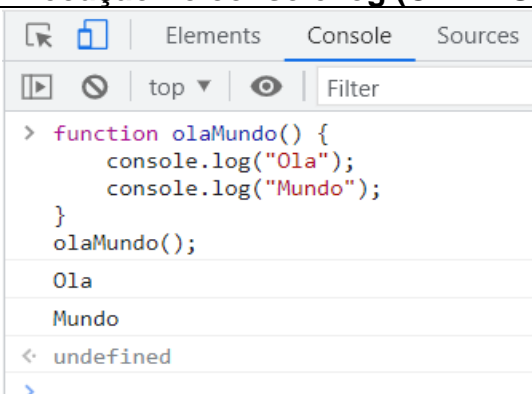
Sintaxe:

```
function nome([param[, param[, ... param]]) {  
  instruções  
}
```

nome: é o nome da função

param: O nome de um argumento a ser passado para a função.

Exemplo de função sem parâmetros:

Exemplo	Execução no console.log (CTRL+SHIFT+J)
<pre>function olaMundo() { console.log("Ola"); console.log("Mundo"); } olaMundo(); // para chamar a função.</pre>	

Observe que a função foi declarada antes de ser chamada. Esta é uma boa prática de programação, pois melhora a legibilidade do código. Caso a função seja declarada depois de ser chamada, ela funcionará também, porém procure declarar as funções antes de chamá-las.

Se você declarar uma variável dentro de um bloco de função, usando *let* ou *var*, ela só ficará visível (e utilizável) dentro do bloco de função. Isso é muito útil porque as variáveis que são usadas dentro de uma função geralmente não são necessárias fora dela.

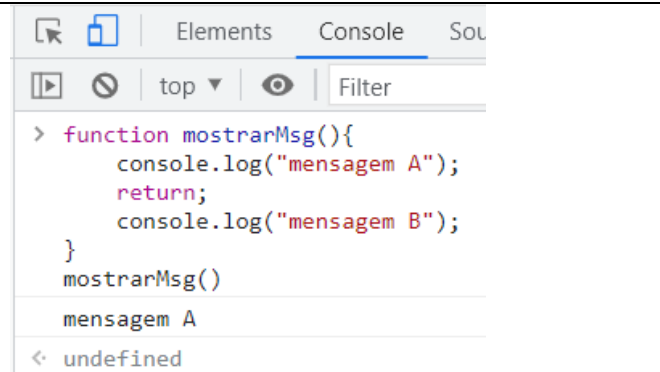
2.1. Return

Podemos utilizar a palavra *return* nas funções. A função termina onde está a palavra *return* e permite armazenar numa variável o resultado da função.

Exemplo:

Exemplo	Execução no console.log (CTRL+SHIFT+J)
---------	--



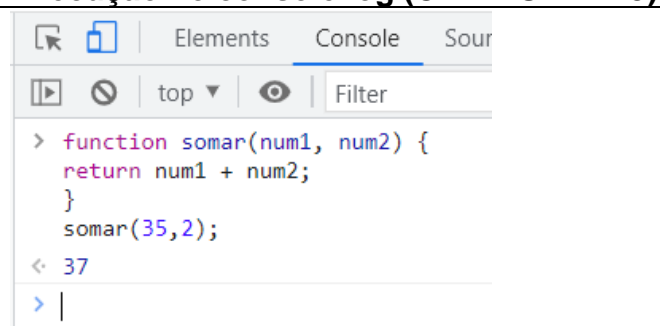
<pre>function mostrarMsg() { console.log("mensagem A"); return; console.log("mensagem B"); } mostrarMsg();</pre>	
--	--

No exemplo acima, ao chamarmos a função **mostrarMsg()** ela exibiu mensagem A. A palavra *return* foi colocada antes da mensagem B, e este texto não será exibido.

2.2. Parâmetros de função

O uso de parâmetros é opcional. Nos exemplos anteriores, não utilizamos parâmetros. Em *JavaScript*, os parâmetros de uma função aparecem em sua declaração. Parâmetros dentro de uma função será tratado como uma variável local. Uma função, cuja definição especifica os parâmetros, deve ser chamada exatamente com estes mesmos parâmetros. Os valores dados durante uma chamada de função são chamados de argumentos. Os argumentos, se houver mais de um, são separados por vírgulas e devem ser passados na mesma ordem dos parâmetros que definimos na declaração da função.

Exemplos:

Exemplo	Execução no console.log (CTRL+SHIFT+J)
<pre>function somar(num1, num2) { //Função somar com parâmetros num1 e num2 return num1 + num2; } somar(35,2); // o resultado exibido será 37. Chamada da função: somar com os argumentos 35 e 2.</pre>	

2.3. Recursividade

Nas aulas de matemática, todos, em algum momento, estudamos fatoriais. Um fatorial é uma função, indicada por um ponto de exclamação em notação matemática. Passamos um inteiro positivo para esta função e seu



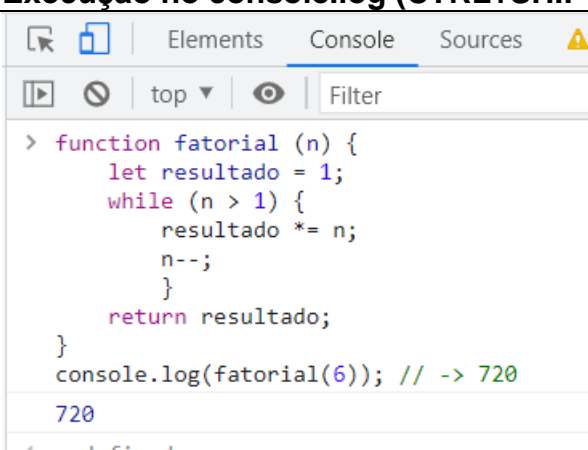
resultado é obtido multiplicando todos os inteiros do número 1 até o número dado como argumento. Um fatorial é definido da seguinte forma:

$$n! = n \times (n-1) \times (n-2) \times \dots \times 2 \times 1$$

Assim, por exemplo, o fatorial de 6 é:

$$6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 720$$

Vamos escrever uma função que calcula o fatorial de um determinado número. Ele receberá o parâmetro *n* e retornará o valor calculado.

Exemplo	Execução no console.log (CTRL+SHIFT+J)
<pre>function fatorial (n) { let resultado = 1; while (n > 1) { resultado *= n; n--; } return resultado; } console.log(fatorial(6));</pre>	

A função fatorial pode ser diferente. Podemos calcular o fatorial do elemento anterior *n-1* multiplicado por *n*. Por exemplo, 6! Seria 5! multiplicado por 6. Vamos utilizar recursividade para esse exemplo:

```
function fatorial (n) {  
  return n > 1 ? n * fatorial(n - 1) : 1;  
}  
  
console.log(fatorial(6)); // -> 720
```

No exemplo acima, utilizamos o **operador ternário**, em vez de utilizarmos a instrução condicional *if*, que é um operador condicional e utiliza três operandos. **Os três operandos são separados uns dos outros por ? (o primeiro do segundo) e : (o segundo do terceiro)**. Verificamos se o argumento *n* é maior que 1, retornamos o resultado da multiplicação do número *n* pelo resultado da chamada fatorial (*n - 1*). Se for menor que 1, o resultado será o valor 1. Funções recursivas devem ser utilizadas com cuidado, e utilizar quando souber o número de “chamadas”, pois **funções recursivas chamam a si próprias** e assim podem entrar em *loop* infinito.



2.4. Função *arrow* (seta)

A função de **seta** (**arrow**) é uma forma mais curta de uma expressão de função. Uma expressão de função de seta é composta de: parênteses contendo ou não parâmetros. Uma seta "=>"; e o corpo da função, que pode ser colocado entre colchetes {}, se o corpo for maior. Se uma função de seta tiver apenas uma instrução e retornar seu valor, podemos omitir a palavra-chave *return*, pois ela será adicionada implicitamente. Por exemplo, a função somar:

```
let somar=(n1,n2) =>n1+n2;  
console.log(somar(2,3)); //5 (5 será o resultado)
```

2.5. Objeto Date ()

O objeto Date é um construtor de datas e horários. Datas e horários são extraídos do sistema operacional do usuário. A data- hora de 1º de janeiro de 1970 às 00:00:00 (h. min, seg) é a hora inicial ou data base para contagem do tempo. Essa referência é conhecida como UnixTime. Essa data é a referência para contagem de tempo passado ou futuro. É expresso em quantidade de milissegundos decorridos desde a data base até o instante considerado (Silva, 2010).

Uma hora em milissegundos é 3.600.000. $1 \text{ h} = 60\text{min} * 60\text{min} * 1000\text{milissegundos}$.

Vamos criar um objeto Date:

```
const agora= new Date();  
alert(agora) //'Thu Jan 19 2023 18:44:38 GMT-0300 (Horário  
Padrão de Brasília)'
```

No exemplo acima, utilizamos o construtor date sem passar argumentos. Observe que ele criou um objeto com a data-hora atual.

2.5.1 Métodos do objeto Date()

O objetoDate possui propriedades, bem como métodos. Na tabela abaixo, apresentamos alguns métodos:

MÉTODO	DESCRIÇÃO/EXEMPLO (Execute esses exemplos no console.)
getDate()	Retorna o dia do mês (1 a 31). Exemplo: <pre>var data = new Date('May 26, 1990') alert(data.getDate()) // irá exibir 26, o dia da data informada</pre>



getDay()	<p>Retorna o dia da semana (0-7) de um objeto Date. Domingo é 0 e segunda-feira é 1. Exemplo:</p> <pre>const data=new Date(); console.log(data.getDay()); // Número equivalente ao dia de hoje let arrayDias=['domingo','segunda-feira','terça-feira','quarta-feira','quinta-feira','sexta-feira','sábado']; let i=data.getDay(); console.log(arrayDias[i]); // Vai exibir o nome do dia da semana.</pre>
getFullYear()	<p>Retorna o ano, com quatro dígitos, de um objeto Date. Exemplo:</p> <pre>var data = new Date('May 26, 1990') alert(data.getFullYear()) // Exibe 1990</pre>
getHours()	<p>Retorna a hora (0-23) de um objeto Date. Exemplo:</p> <pre>var data = new Date('May 26, 1990 04:40:36') alert(data.getHours()) // Exibe 4</pre>
getMonth()	<p>Retorna o mês (0-11) do objeto Date. Exemplo:</p> <pre>var data = new Date('May 26, 1990 04:40:36') alert(data.getMonth()) // Exibe 4, porque inicia em 0 para janeiro, maio equivale ao número 4</pre>
getTime()	<p>Retorna o número de milissegundos transcorridos entre um objeto Date e a hora base 01 de janeiro de 1970. Exemplo:</p> <pre>const dia= new Date('May 26, 1990 04:40:36'); alert(dia.getTime());</pre> <p>Outro exemplo: quantos dias faltam para o Natal:</p> <pre>var hoje, anoCorrente, diaNatal,diferenca,umDia, diasFaltando;// instanciamento das variáveis hoje=new Date(); // criamos a hora e data atual anoCorrente=hoje.getFullYear();//armazenamos o ano corrente numa variável diaNatal=new Date(anoCorrente,11,25);//criamos o dia do natal do ano corrente diferenca=diaNatal.getTime()-hoje.getTime();// armazenamos a diferença em milissegundos entre o natal e o dia de hoje umDia=24*60*60*1000;//armazenamos o tempo de um dia em milissegundos diasFaltando=Math.ceil(diferenca/umDia); // calculamos quantos dias faltam para o natal. Arredondamos para o inteiro superior com o método Math.ceil().</pre>
setTimeout()	<p>Método chama uma função após um determinado tempo (em milissegundos). 1 segundo = 1000 milissegundos. O método é executado apenas uma vez. Sintaxe: <code>setTimeout(função, milissegundos)</code>. Exemplo:</p> <pre><html> <body> <h1>Método setTimeout() </h1> <p>Clique no botão. Aguarde 2 segundos pelo alerta</pre>



	<pre>"Olá Mundo".> <button onclick="myFunction()">Teste</button> //onclick é um evento e ao clicar ele executa a função "myFunction()" <script> // JavaScript executado no HTML (interno) deve estar entre as tags <script> e </script> let timeout; function myFunction() { timeout = setTimeout(alertFunc, 2000); } function alertFunc() { alert("Olá Mundo!"); } </script> </body> </html></pre>
setInterval()	<p>Método chama uma função em intervalos determinados em milissegundos. (1 segundo = 1000 milissegundos). Para interromper é necessário chamar <code>clearInterval()</code> ou fechar a janela. Sintaxe: <code>setInterval(function, milliseconds);</code> Exemplo:</p> <pre><html> <body> <h1>Métodos setInterval() e clearInterval()</h1> <p>Neste exemplo, o método setInterval() executa a função setCor() uma vez a cada 500 milissegundos para alternar entre duas cores de fundo. </p> <button onclick="paraCor()">Parar alternância de cor</button> <script> myInterval = setInterval(setCor, 500); function setCor() { let x = document.body; //elemento que contém o conteúdo do documento. Em documentos com <body>conteúdo, retorna o elemento <body>. x.style.backgroundColor = x.style.backgroundColor == "gray" ? "white" : "gray"; } // style.backgroundColor -> Define uma cor de fundo para um documento function paraCor() { clearInterval(myInterval); //clearInterval() finaliza setInterval() } </script> </body> </html></pre>

Saiba mais

Para conhecer outros métodos de um objeto Date, acesse:
<<https://developer.mozilla.org/en->



US/docs/Web/JavaScript/Reference/Global_Objects/Date>. Acesso em: 6 mar. 2023.

TEMA 3 – BOM (*Browser Object Model*)

O Browser Object Model (BOM) é um conjunto de objetos que permitem a interação entre *JavaScript* e o navegador. Ele fornece acesso às características do navegador, como janelas, histórico, cookies e muito mais.

Com o BOM, é possível criar aplicações web mais interativas e dinâmicas, como janelas de diálogo, pop-ups e outros recursos avançados. No entanto, é importante lembrar que o BOM é específico do navegador e pode não funcionar corretamente em todos os navegadores ou versões.

3.1 Objeto *Window*

O objeto de nível mais elevado do BOM é o *window*. Representa uma janela aberta do navegador. Vamos apresentar as principais propriedades, métodos e eventos do objeto *window* abaixo.

3.1.1 *Closed*

A propriedade *closed* retorna o valor booleano *true* se a janela estiver fechada e *false* se estiver aberta. Exemplo:

```
<html>
<head>
  <script>
    var janelaAberta =
window.open('', '', 'width=400,height=200');
// na linha acima criamos uma janela com a propriedade open.
    function testarJanela() {
      if (janelaAberta.closed) { // se a janela estiver
fechada
        alert('A janela foi fechada');
      } else {
        alert('A janela está aberta');
      }
    }
  </script>
</head>
<body>
  <button type="button" onclick="testarJanela()">Testar
janela</button>
// no eventp onclick chamamos a função "testarJanela()"
```



```
</body>
</html>
```

Saiba mais

Veja no *link* a execução do código acima: <https://n-cpuninter.github.io/fundamentos-de-desenvolvimento-web/html/closed1.html>. Acesso em 6 mar. 2023.

3.1.2 Document

A propriedade **document** é uma referência ao objeto *document* que representa a marcação HTML de um documento apresentado na janela do navegador. É possível acessar todos os elementos da marcação via script.

3.1.3 History

O objeto **history** contém as URLs visitadas pelo usuário (na janela do navegador). Sintaxe:

```
let length = window.history.método;
```

```
let length = history.método;
```

Método	Descrição
back()	Carrega URL anteriormente visitada
forward()	Carrega URL visitada depois da URL atual.
go(n)	Admite número como parâmetro
length()	Quantidade

Fonte: Margarete Klamas, 2023.

Para testar o exemplo abaixo, você deverá abrir a página no navegador, e abrir três *sites* aleatórios, depois usar o botão voltar do navegador e testar os botões Avançar e Avançar 2 janelas. Exemplos:

```
<html>
<body>
  <h1>History</h1>
  <button onclick="history.back()">Voltar</button>
  <button onclick="history.forward()">Avançar</button>
```



```
<button onclick="history.go(2)">Avançar 2 janelas</button>
</body>
</html>
```

Saiba mais

Veja no [link](https://n-cpuninter.github.io/fundamentos-de-desenvolvimento-web/html/history.html) a execução do código acima: <<https://n-cpuninter.github.io/fundamentos-de-desenvolvimento-web/html/history.html>>.

Acesso em: 6 mar. 2023.

3.1.4 Open

O método `open()` abre uma nova janela. Abrir janelas, sem o consentimento do usuário, deve ser evitado. Sintaxe: `window.open(URL, name, specs, replace)`. Os parâmetros são opcionais. Caso não informados, abre uma nova janela em branco.

No exemplo abaixo, vamos abrir o Google por meio de um botão:

```
<html>
<head>
  <script> // abaixo vamos criar a função abreJanela()
  function abreJanela() {

novaJanela=window.open('https://google.com', '_blank', 'width=400,
height=400'); // passamos os parâmetros, URL do Google, _blank
(abrir nova janela), e width (largura) e height (altura).
  }
</script>
</head>
<body>
  <button onclick="abreJanela()">Abrir Janela</button>
  //Chamamos a função abreJanela() no evento onclick do botão

</body>
</html>
```

Saiba mais:

Veja a execução do código em: <<https://n-cpuninter.github.io/fundamentos-de-desenvolvimento-web/html/open.html>>. Acesso em: 6 mar. 2023.

3.1.5 Close

O método `close()` fecha uma janela.

Para o exemplo abaixo, vamos aproveitar o código acima (item 3.1.4) e adicionar a função `fechaJanela()`:



```
<html>
<head>
  <script>
    function abreJanela() {

novaJanela=window.open('https://google.com','_blank','width=400,
height=400');
    }
    function fechaJanela(){ // Criamos a função fecha janela.
      novaJanela.close();
    }
  </script>
</head>
<body>
  <button onclick="abreJanela()">Abrir Janela</button>
  <button onclick="fechaJanela()">Fechar Janela</button> //
chamamos a função fechaJanela()
</body>
</html>
```

Saiba mais:

Veja a execução do código em: <https://n-cpuninter.github.io/fundamentos-de-desenvolvimento-web/html/v_open_close.html>. Acesso em: 6 mar. 2023.

3.1.6 Onload

O evento onload() ocorre quando uma página termina de ser carregada.

Exemplo:

```
<html>
<head>
  <script>
    window.onload = function() {
      document.forms[0].elements[0].value = 'Anna';
      document.forms[0].elements[1].value = 'Silva';
    }
  </script>
</head>
<body>
  <form action="" method="get">
    <input type="text" />
    <input type="text" />
  </form>
</body>
</html>
```

Saiba mais:

Veja a execução do código em: <https://n-cpuninter.github.io/fundamentos-de-desenvolvimento-web/html/v_onload.html>. Acesso em: 6 mar. 2023.

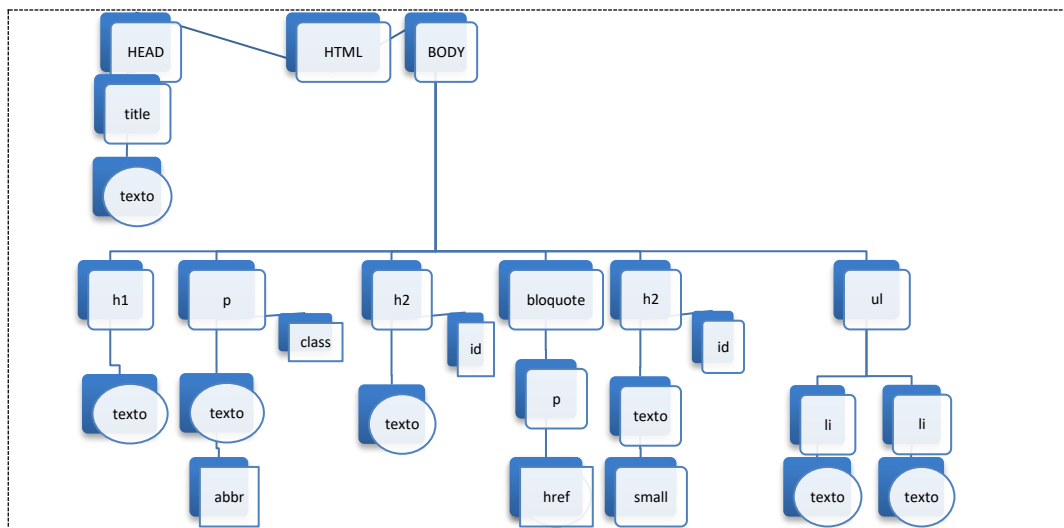


TEMA 4 – DOM (*Document Object Model*)

A definição do DOM segundo o W3C¹: “O DOM – Document Object Model é uma interface independente de plataforma e linguagem que permite aos programas e scripts acessar e atualizar dinamicamente a estrutura, o conteúdo e a estilização de documentos”.

O DOM HTML é uma representação da estrutura do documento HTML. O diagrama representativo do DOM é do tipo árvore genealógica. O DOM padroniza a estrutura de documentos HTML, simplificando a tarefa de acessar e manipular esses documentos.

REPRESENTAÇÃO DO DOM



Fonte: Silva, 2010.

4.1 Objeto Document

O objeto *document* representa um documento aberto no navegador. Assim, quando abrimos um documento HTML em um navegador, é criado um objeto documento. Esse objeto é um objeto *window* e pode ser acessado com a sintaxe *window.document*. Desta maneira, podemos acessar todos os elementos HTML de uma página HTML.

4.2 *etElement**

¹ World Wide Web Consortium.



O método **`getElementById()`** acessa o elemento DOM identificando o **id**. ID é identificador único, exclusivo no documento.

Exemplo de **`getElementById`**:

```
<html>
<head>
  <meta charset="UTF-8">
  <title>Document</title>
  <script>//Se utili
    window.onload = function() {
      var Um = document.getElementById("def");//Aqui selecionamos
o id "def"
      var Dois = document.getElementById("tipo");//Aqui
selecionamos o id "tipo"
      Um.style.backgroundColor="#0cc";//Aqui atribuímos uma
cor de fundo
      Dois.style.border="2px solid #0000ff";//Aqui atribuímos
uma borda
    } </script>
</head>
<body>
  <h2>Introdução</h2>
  <p>Lorem ipsum </p>
  <h2 id="def">Definição</h2> <!--Aqui colocamos o id="def"-->
  <p>Lorem ipsum?</p>
  <h2 id="tipo">DOM</h2> <!--Aqui colocamos o id="tipo"-->
  <ul>
    <li>DOM para HTML</li>
    <li>DOM para XML</li>
  </ul>
</body>
</html>
```

No exemplo acima, criamos duas variáveis “Um” e “Dois” que se referem ao elemento `<h2>` que marcam os títulos “Definição” e “DOM”. No exemplo, colocamos um sombreamento e borda, nos títulos.

O **`getElementsByClassName()`** retorna uma lista com todos os elementos que possuem o nome da classe dada.

Exemplo:

```
Exemplo: let lista = document.getElementsByClassName("item")
```

O **`getElementsByTagName`** retorna uma lista com todos os elementos da tag informada.

```
Exemplo: let lista = document.getElementsByTagName("p")
```



4.3 *element.style*

A propriedade *style* dos elementos do DOM permite que se definam regras de estilo. Sintaxe: `el.style.propriedade="valor da propriedade"`. No item 4.2, apresentamos exemplos com aplicação dessa propriedade.

Em *JavaScript*, as propriedades CSS devem ser escritas em *camelCase*, como nos exemplos abaixo:

Sintaxe JavaScript para CSS

CSS	JavaScript
background-color	backgroundColor
z-index	zIndex
text-indent	textIndent
padding-left	paddingLeft

4.4 *querySelector(seletor)*

O método `querySelector()` permite acessar o primeiro elemento correspondente ao seletor informado. O seletor pode ser uma tag, um id ou uma classe. São os mesmo que estudamos no CSS. Exemplos:
`let tituloH1=document.querySelector("h1").`

4.5 *querySelectorAll(seletor)*

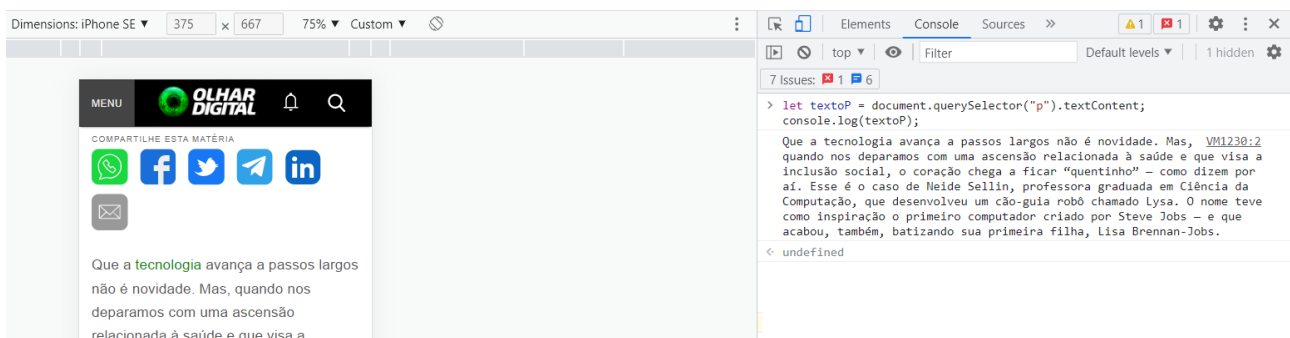
Permite acessar todos os elementos que correspondem ao seletor.
Exemplo:

```
document.querySelectorAll("p").
```

4.6 *TextContent*

Permite acessar o texto digitado entre as tags. Por exemplo, abrimos o *site* do olhar digital, o console (CTRL+SHIFT+J) e digitamos o código abaixo:

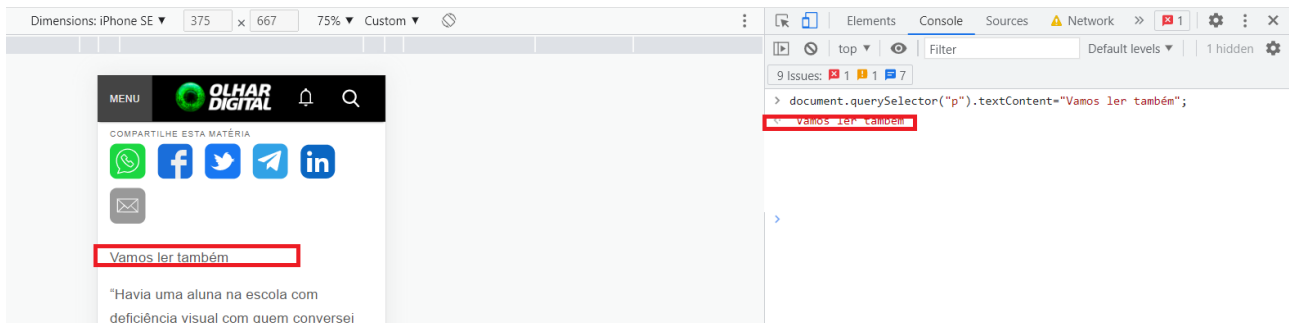
```
let textoP = document.querySelector("p").textContent;  
console.log(textoP);
```





Ele irá exibir o conteúdo da primeira tag p do *site*. Também permite alterar o conteúdo da tag. Essa alteração é somente na máquina do usuário. Exemplo:

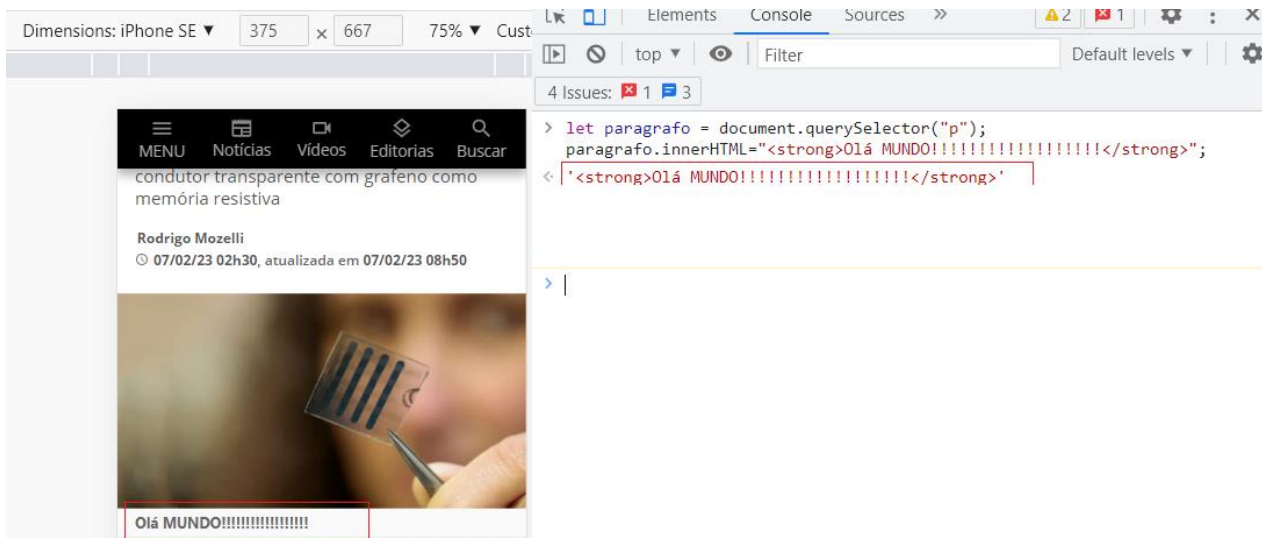
```
document.querySelector("p").textContent="Vamos ler também";
```



4.7 innerHTML

Permite adicionar HTML dentro do HTML. Exemplo:

```
let paragrafo = document.querySelector("p");
paragrafo.innerHTML="<strong>Olá MUNDO!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!</strong>";
```



TEMA 5 – EVENTOS

Os eventos podem ocorrer advindos da interação do usuário, como o clique em um *link*, ou passar o *mouse* sobre, ou advindos do navegador, como carregar uma página. Evento é qualquer interação com um elemento HTML, clicar, mover, soltar etc.



- São disponíveis eventos no mouse. Estes ocorrem quando o usuário interage, usando o ponteiro do mouse.
- Eventos do teclado ocorrem quando o usuário interage utilizando as teclas do teclado.
- Eventos HTML ocorrem quando há alterações na janela do navegador.

Exemplos de Eventos

Evento	Descrição
<i>click</i>	Ocorre quando o usuário pressiona o botão esquerdo do mouse.
<i>dblclick</i>	Ocorre quando o usuário pressiona duas vezes o botão esquerdo do mouse;
<i>mousedown</i>	Ocorre quando o usuário pressiona qualquer um dos botões do mouse.
<i>mouseover</i>	Ocorre quando o usuário solta qualquer um dos botões do mouse.
<i>mouseout</i>	Ocorre quando o usuário desloca o ponteiro do mouse de onde estava.
<i>mousemove</i>	Ocorre quando o usuário move o ponteiro do mouse.
<i>load</i>	Ocorre quando há o carregamento da página.
<i>unload</i>	Ocorre quando há o fechamento de uma janela.
<i>scroll</i>	Ocorre quando o elemento possui barra de rolagem.
<i>focus</i>	Ocorre quando o elemento recebe foco.

Saiba mais:

Para conhecer mais sobre eventos, clique em:

1. <https://developer.mozilla.org/pt-BR/docs/Learn/JavaScript/Building_blocks/Events.
2. <https://www.w3schools.com/js/js_html_dom_events.asp.

5.1 *eventListener*

O evento *eventListener* possui os métodos *addEventListener* e *removeEventListener* que nos permitem anexar um manipulador de eventos a um elemento sem sobrescrever os eventos anteriores.

Sintaxe:

```
element.addEventListener(event, function, useCapture);
```

O parâmetro *event* é o tipo de evento, exemplo: *click* ou *mouseout*.

O parâmetro *function* é a função que queremos chamar.

O parâmetro *useCapture* é um valor booleano que especifica se é para usar captura de eventos. É opcional. Acompanhe o exemplo nos vídeos:



Baixe o código: <https://github.com/n-cpuninter/fundamentos-de-desenvolvimento-web/blob/main/txt-codigos/eventlistener.txt>

Parte 1 do vídeo: https://n-cpuninter.github.io/fundamentos-de-desenvolvimento-web/html/v_addeventlistenera.html

Parte 2 do vídeo: https://n-cpuninter.github.io/fundamentos-de-desenvolvimento-web/html/v_addeventlistenerb.html

FINALIZANDO

Nesta abordagem, estudamos um pouco mais de *JavaScript*. Usamos a expressão “um pouco mais” porque *JavaScript* é muito amplo e tem muito a ser estudado.

Recomendamos que você reproduza os exemplos aqui apresentados, digite-os para treinar a sintaxe. *JavaScript* é uma linguagem muito agradável e importante para você aprender e, nesta fase, você deu os primeiros passos nessa linguagem.

Agora poderemos interagir um pouco mais com os usuários!



REFERÊNCIAS

FLANAGAN, D. **JavaScript**. o guia definitivo. – 6. ed. – Porto Alegre: Bookman, 2013.

Mozilla Developer Network. **JavaScript**. Disponível em: < [https:// developer.mozilla.org/ en-US/](https://developer.mozilla.org/en-US/) >. Acesso em: 6 mar. 2023.

OLIVEIRA, C.; ZANETTO, HAP. **JavaScript descomplicado**: programação para a Web, IOT e dispositivos móveis. – São Paulo: Érica, 2020.

SILVA, M. S. **JavaScript - Guia do programador**. São Paulo: Novatec Editora, 2010.

W3C. **JavaScript**. Disponível em: <[http:// www.w3.org](http://www.w3.org)>. Acesso em: 6 mar. 2023.