



Estudos – JavaScript

Guia de Aprendizado – JavaScript e suas tecnologias

Nome: Carlos Eduardo Cerqueira Leite

Cargo: Desenvolvedor de Software

São José do Rio Preto, São Paulo, Brasil

Sumário

1	Comandos.....	6
1.1	Comandos para conversão de tipos.....	6
1.2	Comandos console.....	6
1.3	Comandos window	6
1.4	Comandos document	6
1.5	Comandos para adicionar eventos.....	7
1.6	Comandos para manipulação de dados.....	7
1.7	Comandos para manipulação de elementos	8
1.8	Comandos para manipulação de Arrays	8
2	O que são variáveis	9
2.1	Conversão de dados em variáveis	9
2.2	Concatenação de dados	10
2.3	Interpolação de dados.....	11
2.4	Manipulando Dados através de Comandos	11
3	Operadores	14
3.1	Operadores Aritméticos.....	15
3.2	Operadores de Atribuição	15
3.2.1	Operadores de Incremento.....	17
3.3	Operadores Relacionais.....	18
3.4	Operadores Lógicos.....	19
3.5	Operadores Ternários	20
4	Estruturas Condicionais	22
4.1	Condição Simples	22
4.2	Condição Composta.....	22
4.3	Switch Case	23
5	DOM (Document Object Model)	25
5.1	A Árvore DOM.....	25
5.2	Manipulando o DOM com JavaScript.....	26
5.2.1	Alterações de estilo por meio de JavaScript.....	30
5.2.2	.innerText e .innerHTML	33
5.3	Eventos DOM.....	36
5.3.1	Funções.....	36
6	Estruturas de Repetição.....	39
6.1	While	39

6.2	Do While.....	39
6.3	Quando optar por While ou Do While?	40
6.4	For.....	40
7	Depuração de Código	42
8	Variáveis Compostas (Arrays (ou Vetores))	48
8.1	Exibição de elementos de Arrays através do for	49
8.2	Manipulação de Arrays.....	50
9	Objetos.....	51
10	Funções.....	53
10.1	Hoisting.....	54
11	Conclusão.....	56

Sumário de Figuras

Figura 1 – Algoritmo de Operações Básicas	9
Figura 2 – Retorno do window.alert() no navegador Google Chrome	10
Figura 3 – Utilização de String(n)	10
Figura 4 - Exemplo de Concatenação para exibição de valores de variáveis .	11
Figura 5 – Exibição de dados utilizando a Interpolação	11
Figura 6 – Utilização de comandos para manipular dados.....	12
Figura 7 – Retorno do document.write().....	12
Figura 8 – Utilização de toFixed().....	12
Figura 9 – Utilização de replace() para substituir elementos.....	13
Figura 10 – Aplicando prefixo monetário ao valor da variável	13
Figura 11 – Operadores Aritméticos alterando o valor da mesma variável	16
Figura 12 - Operadores de Atribuição alterando o valor da mesma variável....	16
Figura 13 – Aplicação de n++ e n—	17
Figura 14 – Aplicação de ++n e –n.....	18
Figura 15 – Aplicação e Retorno de Operadores Relacionais.....	18
Figura 16 – Comparação entre == e ===	19
Figura 17 – Execução de Comparações Lógicas	20
Figura 18 – Aplicação de Operador Ternário	21
Figura 19 – Atribuindo valor a uma variável através de validação com Operadores Ternários	21
Figura 20 - Condição Simples	22

Figura 21 - Condição Composta.....	22
Figura 22 - Estrutura Condicional Switch	23
Figura 23 – Árvore DOM	25
Figura 24 – Método de Seleção por Marca	26
Figura 25 – Retorno do Método de Seleção por Marca.....	27
Figura 26 – Método de Seleção por ID.....	27
Figura 27 – Retorno do Método de Seleção por ID	28
Figura 28 – Método de Seleção por Nome.....	28
Figura 29 – Método de Seleção por Classe	29
Figura 30 – Aplicação e Retorno de querySelector().....	29
Figura 31 – Aplicação de querySelectorAll()	30
Figura 32 – Retorno da Aplicação de querySelectorAll().....	30
Figura 33 – Alteração de background com JavaScript	31
Figura 34 – Retorno da Alteração de background com JavaScript	31
Figura 35 – Produto da Alteração de background com JavaScript.....	32
Figura 36 – Alterações com mais de 2 palavras.....	32
Figura 37 – Retorno dessas Alterações com mais de 2 palavras.....	32
Figura 38 – Produto dessas Alterações com mais de 2 palavras.....	33
Figura 39 - Utilização de .innerText.....	33
Figura 40 - Retorno de Utilização de .innerText	33
Figura 41 – Utilização de .innerHTML	34
Figura 42 – Retorno de Utilização de .innerHTML	34
Figura 43 – Utilização de .innerText em document.write()	34
Figura 44 – Retorno de Utilização de .innerText em document.write().....	35
Figura 45 – Utilização de .innerHTML em document.write().....	35
Figura 46 – Retorno de Utilização de .innerHTML em document.write()	36
Figura 47 - Aplicação de Função.....	37
Figura 48 - Aplicação de várias funções ao mesmo elemento	37
Figura 49 - Utilização de Event Listener.....	38
Figura 50 - Estrutura de Repetição While	39
Figura 51 - Retorno da Estrutura de Repetição While	39
Figura 52 - Estrutura de Repetição Do While	40
Figura 53 - Retorno da Estrutura de Repetição Do While	40
Figura 54 - Estrutura de Repetição For	41
Figura 55 - Retorno da Estrutura de Repetição For	41

Figura 56 - Abrindo a Ferramenta de Debug.....	42
Figura 57 - Seleção da Ferramenta de Depuração	42
Figura 58 - Seleção de opções de depuração.....	43
Figura 59 - Abertura do Menu Lateral e Console de Depuração	43
Figura 60 - Opção de Adição de Breakpoints.....	44
Figura 61 - Seleção do trecho de código a ser depurado.....	44
Figura 62 - Adição de variável ao campo Watch	44
Figura 63 - Variável adicionada no campo Watch	45
Figura 64 - Código em andamento.....	45
Figura 65 - Utilização da opção Step Over (F10)	46
Figura 66 - Passo seguinte na depuração do código	47
Figura 67 - Variável Composta Homogênea	48
Figura 68 - Variável Composta Heterogênea	48
Figura 69 - Exibição de Elementos de um Array através do For	49
Figura 70 - Utilização de sintaxe especial do For	50
Figura 71 - Exemplo de Objeto.....	51
Figura 72 - Acessando Dado de um objeto a partir de sua Propriedade	52
Figura 73 - Alterando Dados em funções presentes em Objetos	52
Figura 74 - Exemplo de Função	53
Figura 75 - Exemplo de Função com 2 ou mais parâmetros	53
Figura 76 - Exemplo de retorno não esperado por falta de parâmetros	54
Figura 77 - Exemplo de função definida como valor de variável	54
Figura 78 - Explicação de Hoisting.....	55

1 Comandos

A cadeia de comandos no JavaScript consiste em indicar o elemento e, separando por um '.', indicar a ação necessária.

Existem também comandos mais recentes é necessário somente indicar a ação e a própria linguagem averigua a conversão ideal.

A utilização desses comandos é sucedida pelos parênteses, neles é indicado o argumento do comando.

Neste capítulo iremos nos referir as variáveis pelo nome genérico **n** e aqui indicaremos comandos para tratamento de dados.

1.1 Comandos para conversão de tipos

Number(n) – Converte **n** para o tipo **Number**, averigua se deve converter para **Float** ou **Inteiro**;

Number.parseInt(n) – Converte **n** especificamente para o tipo **Inteiro**;

Number.parseFloat(n) – Converte **n** especificamente para o tipo **Float**;

String(n) – Converte **n** para o tipo **String**;

n.toString() – Converte **n** para o tipo **String**;

1.2 Comandos console

console.log() - Utilizado para exibir dados no console do navegador e/ou no console do editor de código.

1.3 Comandos window

window.prompt() – Exibe uma caixa de diálogo com uma mensagem opcional solicitando ao usuário a entrada de algum texto;

window.alert() – Exibe uma caixa de diálogo de aviso com o conteúdo opcionalmente especificado e um botão OK;

1.4 Comandos document

document.write() – Insere uma String no documento.

`document.getElementsByTagName('tag')` - Seleciona elementos pela tag, possui a **tag** do elemento como argumento.

`document.getElementById('id')` - Seleciona elementos pelo id, possui o atributo **id** do elemento como argumento.

`document.getElementsByName('name')` - Seleciona elementos pelo nome, possui o atributo **name** do elemento como argumento.

`document.getElementsByClassName('class')` - Seleciona elementos pela classe, possui o atributo **class** do elemento como argumento.

`document.querySelector()` - Realiza uma busca e retorna o primeiro elemento encontrado, independente de qual seja o atributo ou elemento.

`document.querySelectorAll()` - Tem como objetivo selecionar todos os elementos encontrados na busca e então indicar a próxima ação.

`document.createElement()` - Cria um elemento HTML no DOM.

1.5 Comandos para adicionar eventos

`n.addEventListener('evento', função)` - Indica um evento que deverá ocorrer após determinada ação (**click**, **mouseenter**, **mouseout** e etc). É possível também declarar diretamente a função dessa forma: `n.addEventListener('evento', função() { })`

1.6 Comandos para manipulação de dados

`${n}` - Marcação de Placeholder para interpolar uma variável;

`n.length` - Exibe a quantidade de elementos em **n**, no caso de uma String, exibe a quantidade de caracteres;

`n.toUpperCase()` - Converte todos os caracteres de **n** para letras minúsculas;

`n.toLowerCase()` - Converte todos os caracteres de **n** para letras minúsculas;

`n.toFixed()` - Define quantas casas depois da vírgula serão exibidas;

`n.replace('original', 'novo')` - Substitui um elemento por outro, sua sintaxe exige 2 argumentos: o elemento original e o elemento que irá substituí-lo;

`n.toLocaleString()` - Indica que se trata de uma String localizada, pertencente a determinada parte do mundo;

`n.innerText` - Insere um texto sem formatação pré-definida, ignorando tags HTML.

`n.innerHTML` - Insere um texto com a formatação do elemento original, inclusive tags HTML.

1.7 Comandos para manipulação de elementos

`n.setAttribute('atributo', 'valor do atributo')` – Define o valor de um atributo ao elemento representado pela variável `n`.

`n.appendChild(x)` – Anexa um elemento filho à `n`, sendo `x` o elemento filho.

`n.getFullYear()` – Busca o ano através do horário local, essa lógica se aplica a outras possíveis unidades de medida de tempo.

`n.focus()` – Mantém o foco no elemento representado por `n`.

1.8 Comandos para manipulação de Arrays

`n[índice] = valor` – Ao indicar o índice, é possível selecionar em que posição do Array deseja que o valor seja armazenado.

`n.push(valor)` – Insere o valor diretamente na última posição vazia do Array.

`n.length` – Exibe o comprimento do Array, a quantidade de elementos presentes.

`n.sort()` – Considerando um Array com dados do tipo Number, reorganiza os elementos e os exibe em ordem crescente.

`n.indexOf(valor)` – Exibe o índice do Array onde se localiza o valor especificado. Caso o valor não seja encontrado no Array, por padrão o valor `-1` é retornado.

2 O que são variáveis

Uma variável é um espaço na memória do computador destinado a um dado cujo valor pode ser alterado durante a execução do algoritmo. Para funcionar corretamente, as variáveis precisam ser definidas por nomes e tipos. Sendo eles:

- **Number:** Para números, seja inteiro ou flutuante.
- **String:** Para sequências de caracteres.
- **Boolean:** Para valores verdadeiro ou falso.
- **Object:** Para coleções de dados ou instâncias de construtores.
- **Undefined:** Para variáveis não inicializadas.
- **Null:** Representa a ausência de valor.
- **Function:** São blocos de código que podem ser reutilizados e executados quando necessário.

O JavaScript é dinamicamente tipado. Isso significa que o tipo de dado de uma variável pode mudar durante a execução do programa.

2.1 Conversão de dados em variáveis

Alguns comandos por padrão no JavaScript interpretam os dados neles inseridos como sendo do tipo **String**. Dependendo da aplicação, torna-se necessário fazer a conversão desse dado para outro tipo, conforme mostra a figura a seguir:

```
<script>
var n1 = Number(window.prompt('Digite um número:'));
var n2 = Number(window.prompt('Digite outro número:'));

soma = n1 + n2;
sub = n1 - n2;
mult = n1 * n2;
div = n1 / n2;

window.alert('Você digitou os números ' + n1 + ' e ' + n2 + '.\n A soma desses números é: ' + soma + '.\n A subtração desses números é: ' + sub + '.\n A multiplicação desses números é: ' + mult + '.\n A divisão desses números é: ' + div + '.');
</script>
```

Figura 1 – Algoritmo de Operações Básicas

Nesse Algoritmo, foi utilizado o comando **Number(n)** (sendo n um nome genérico para o valor da variável) para realizar a conversão dos dados coletados pelo **window.prompt()** (Neste objeto, os dados por padrão são tratados como sendo do tipo **String**). A figura a seguir exibe o que este **window.alert()** retorna:

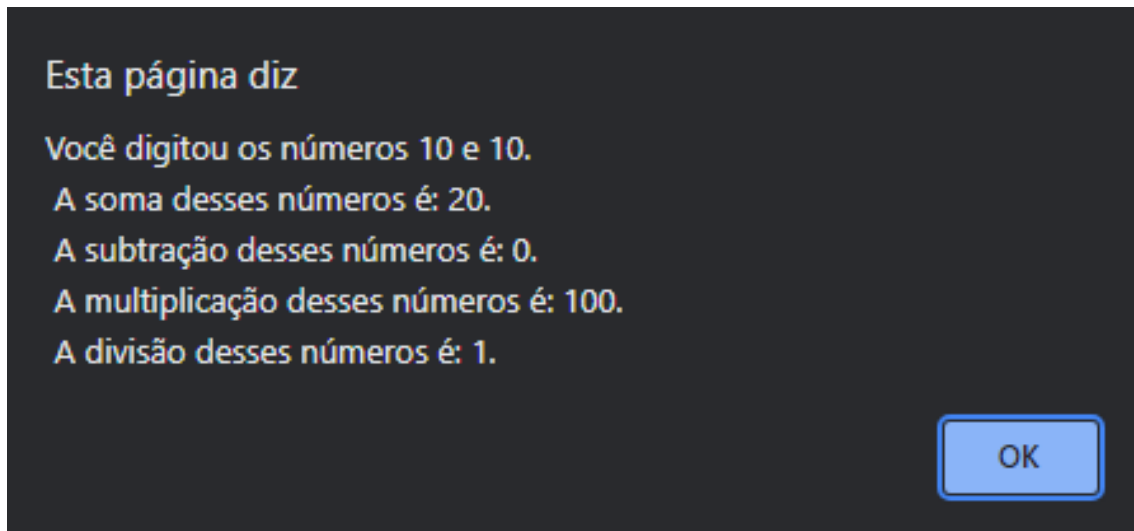


Figura 2 – Retorno do `window.alert()` no navegador Google Chrome

Este comando detecta se o dado inserido é do tipo `Inteiro` ou do `Float`, é também possível forçar a conversão específica para um desses tipos de dados através dos comandos `Number.parseInt()` e `Number.parseFloat()`.

É também possível realizar o inverso (converter `Number` para `String`) através do comando mais abrangente `String(n)` ou através do comando `n.toString()`.
Segue exemplo:

```
<script>
var n1 = Number(window.prompt('Digite um número:'));
var n2 = Number(window.prompt('Digite outro número: '));

soma = n1 + n2;
sub = n1 - n2;
mult = n1 * n2;
div = n1 / n2;

window.alert('Você digitou os números ' + String(n1) + ' e ' + String(n2) + '.\n A soma desses números é: ' + soma + '.\n A
subtração desses números é: ' + sub + '.\n A multiplicação desses números é: ' + mult + '.\n A divisão desses números é: ' +
div + '.');
</script>
```

Figura 3 – Utilização de `String(n)`

2.2 Concatenação de dados

Nas figuras 1 e 2, o método utilizado para exibir o valor das variáveis junto ao texto de orientação foi o da concatenação. Este método consiste em exibir a variável através do operador `+` conforme indica a figura a seguir:

```
window.alert('Você digitou os números ' + String(n1) + ' e ' + String(n2) +  
'.\n A soma desses números é: ' + soma + '.\n A subtração desses números é:  
' + sub + '.\n A multiplicação desses números é: ' + mult + '.\n A divisão  
desses números é: ' + div + '.');
```

Figura 4 - Exemplo de Concatenação para exibição de valores de variáveis

A seguir será apresentado outro método, o da Interpolação de dados.

2.3 Interpolação de dados

Neste método, a sintaxe consiste em utilizar a crase para envolver todo o texto que será exibido, além de utilizar `${n}` para exibir o valor da variável na própria String, sem a necessidade de cortá-la em partes como torna-se necessário na concatenação.

Chamamos o elemento `${n}` de **Placeholder** e uma String envolvida por crase chamamos de **Template String**

```
<script>  
var n1 = Number(window.prompt('Digite um número:'));  
var n2 = Number(window.prompt('Digite outro número: '));  
  
soma = n1 + n2;  
sub = n1 - n2;  
mult = n1 * n2;  
div = n1 / n2;  
  
window.alert(`Você digitou os números ${String(n1)} e ${String(n2)}.  
\n A soma desses números é: ${soma}.\n A subtração desses números  
é: ${sub}.\n A multiplicação desses números é: ${mult}.\n A divisão  
desses números é: ${div}.`);  
</script>
```

Figura 5 – Exibição de dados utilizando a Interpolação

Note que neste método todo o conteúdo está envolvido pela crase e o `+` não foi utilizado, tornando o código mais legível e otimizado.

2.4 Manipulando Dados através de Comandos

É possível também fazer uso de comandos para manipular os dados que são atribuídos aos valores das variáveis, conforme mostra o código a seguir:

```
<script>
  var nome = window.prompt('Qual seu nome?');
  document.write(`Olá ${nome}! Seu nome tem ${nome.length} letras.<br>`);
  document.write(`Seu nome em letras maiúsculas é ${nome.toUpperCase()}.<br>`);
  document.write(`Seu nome em letras minúsculas é ${nome.toLowerCase()}.<br>`);
</script>
```

Figura 6 – Utilização de comandos para manipular dados

No exemplo anterior foi utilizado os comandos `toUpperCase ()` para converter os caracteres da String para letra maiúscula e `toLowerCase ()` para converter os caracteres da String para letra minúscula. Foi utilizado também o comando `length` para exibir a quantidade de caracteres presentes na String.

A seguir, o retorno desse script:

Olá Carlos! Seu nome tem 6 letras.
Seu nome em letras maiúsculas é CARLOS.
Seu nome em letras minúsculas é carlos.

Figura 7 – Retorno do `document.write()`

É possível aplicar este tratamento também para números, utilizaremos como exemplo o comando `toFixed()`, através dele é possível definir quantas casas depois da vírgula serão exibidas. Exemplo:

```
> var n1 = 1545.5
undefined
> n1
1545.5
> n1.toFixed(2)
'1545.50'
> n1.toFixed(4)
'1545.5000'
> █
```

Figura 8 – Utilização de `toFixed()`

Indo mais além, é possível ampliar a cadeia de comandos e não se limitar a apenas um. Como exemplo substituiremos o “.” por “,”.

```
> var n1 = 1545.5
undefined
> n1
1545.5
> n1.toFixed(2)
'1545.50'
> n1.toFixed(4)
'1545.5000'
> n1.toFixed(2).replace('.', ',')
'1545,50'
>
```

Figura 9 – Utilização de replace() para substituir elementos

Outro exemplo é a utilização de comandos para adicionar o prefixo monetário de determinada moeda, conforme mostra o exemplo:

```
> var n1 = 1545.5
undefined
> n1
1545.5
> n1.toFixed(2)
'1545.50'
> n1.toFixed(4)
'1545.5000'
> n1.toFixed(2).replace('.', ',')
'1545,50'
> n1.toLocaleString('pt-BR', {style: 'currency', currency: 'BRL'})
'R$ 1.545,50'
> n1.toLocaleString('pt-BR', {style: 'currency', currency: 'USD'})
'US$ 1.545,50'
> n1.toLocaleString('pt-BR', {style: 'currency', currency: 'EUR'})
'€ 1.545,50'
>
```

Figura 10 – Aplicando prefixo monetário ao valor da variável

No exemplo acima foi utilizado o comando `toLocaleString()` para indicar que se trata de uma String localizada, pertencente a determinada parte do mundo, no argumento indicamos o Brasil como referência (pt-BR) e em seguida inserimos um objeto (o conteúdo entre {}) ao argumento para indicar qual moeda desejamos utilizar, no caso utilizamos respectivamente BRL (Real), USD (Dólar) e EUR (Euro).

3 Operadores

Neste capítulo iremos abordar os diferentes tipos de operadores presentes no JavaScript e também em outras linguagens, sendo eles os Operadores:

- **Aritméticos:** Utilizados para cálculos básicos:
 - **+** (Adição);
 - **-** (Subtração);
 - ***** (Multiplicação);
 - **/** (Divisão);
 - **%** (Módulo ou Resto);
 - ****** (Potência).
- **Atribuição:** Utilizados para atribuir valores a variáveis enquanto realiza também um cálculo aritmético:
 - **=** (Atribuição Simples);
 - **+=** (Atribuição Aditiva);
 - **-=** (Atribuição Subtrativa);
 - ***=** (Atribuição Multiplicativa);
 - **/=** (Atribuição de Divisão);
 - ****=** (Atribuição de Potência);
 - **%=** (Atribuição de Módulo).
- **Relacionais:** Utilizados para comparar valores:
 - **>** (Maior que);
 - **<** (Menor que);
 - **>=** (Maior ou igual que);
 - **<=** (Menor ou igual que);
 - **==** (Igual a);
 - **===** (Idêntico a);
 - **!=** (Diferente de);
 - **!==** (Não Idêntico a).
- **Lógicos:** Também utilizados para comparar valores:
 - **!** (Negação);
 - **&&** (Conjunção ou 'E' lógico);
 - **||** (Disjunção ou 'OU' lógico).

- **Ternários:** Utilizado para efetuar um teste lógico e possui 3 operandos:
 - **?** (**Se** ou **If**);
 - **:** (**Senão** ou **Else**).

Quando se faz uso de Operadores, é importante ressaltar que existe uma ordem de precedência entre eles, devemos calcular primeiro os Operadores Aritméticos (os Operadores de Atribuição devem ser considerados como Aritméticos na ordem de precedência, pois sua função é encurtar um cálculo aritmético), depois os Operadores Relacionais, então os Operadores Lógicos e por fim os Operadores Ternários.

Este conceito será melhor detalhado nos capítulos seguintes.

3.1 Operadores Aritméticos

Assim como na Matemática, na Programação também deve ser considerada a ordem de precedência ao efetuar um cálculo. A ordem de precedência dos Operadores Aritméticos é:

Operador	Descrição
()	Tudo entre () passa a ser prioridade
**	Potência
* / %	Multiplicação, Divisão e Resto
+ -	Adição e Subtração

Novamente, assim como na Matemática, em casos onde os operadores possuem a mesma ordem de precedência, a prioridade pertence àquele vier primeiro, da esquerda para a direita.

3.2 Operadores de Atribuição

Estes operadores tem como funcionalidade atribuir valores a uma variável enquanto realiza também algum cálculo aritmético, a seguir destacaremos suas funcionalidades:

Operador	Descrição
=	Atribuição Simples
+=	Atribuição Aditiva
-=	Atribuição Subtrativa
*=	Atribuição Multiplicativa
/=	Atribuição de Divisão
**=	Atribuição de Potência
%=	Atribuição de Módulo

Os Operadores além da Atribuição Simples tem como principal utilidade simplificar operações onde o valor da mesma variável é alterado em uma operação, segue exemplo:

```
> var n = 3
undefined
> n = n + 4
7
> n = n - 5
2
> n = n * 4
8
> n = n / 2
4
> n = n ** 2
16
> n = n % 5
1
>
```

Figura 11 – Operadores Aritméticos alterando o valor da mesma variável

Note que no exemplo anterior as operações envolveram uma única variável e todos os Operadores Aritméticos foram utilizados para atribuir um novo valor a essa variável. Este processo pode ser otimizado utilizando os diferentes Operadores de Atribuição, segue exemplo:

```
> var n = 3
undefined
> n += 4
7
> n -= 5
2
> n *= 4
8
> n /= 2
4
> n **= 2
16
> n %= 5
1
>
```

Figura 12 - Operadores de Atribuição alterando o valor da mesma variável

As mesmas operações foram efetuadas, porém através de um código mais limpo e menos repetitivo.

3.2.1 Operadores de Incremento

Ainda no princípio dos Operadores de Atribuição, existem os Operadores de Incremento (que possuem duas formas, a pós-fixada e a pré-fixada), que são uma outra simplificação para outro caso muito utilizado, o acréscimo ou decréscimo de 1 número. Segue exemplo do modelo pós-fixado:

```
> var n = 10
undefined
> n++
10
> n
11
> n++
11
> n
12
> n--
12
> n
11
> n--
11
> n
10
>
```

Figura 13 – Aplicação de `n++` e `n--`

Note que após fazer uso do `n++`, o valor retornado ainda é 10, mesmo tendo sido adicionado 1 número a mais à variável, e logo depois ao solicitar o valor de `n`, é comprovado que este único número foi adicionado (a mesma lógica se aplica ao `n--`). Isso se deve a lógica de funcionamento dos Operadores de Incremento.

No modelo pós-fixado (`n++` e `n--`), ele atualiza o valor da variável, porém utiliza o valor original.

No modelo pré-fixado (`++n` e `--n`), ele utiliza o valor atualizado e em seguida atualiza o valor da variável. Segue exemplo:

```

> var n = 10
undefined
> ++n
11
> n
11
> ++n
12
> n
12
> --n
11
> n
11
> --n
10
> n
10
> 

```

Figura 14 – Aplicação de ++n e --n

3.3 Operadores Relacionais

Os Operadores Relacionais tem como utilidade efetuar comparações entre 2 valores, retornando como resultado valores **booleanos** (**true** ou **false**). Segue exemplo:

```

> 5 > 2
true
> 7 < 4
false
> 8 >= 8
true
> 9 <= 7
false
> 5 == 5
true
> '5' === 5
false
> 4 != 4
false
> '4' !== 4
true
> 

```

Figura 15 – Aplicação e Retorno de Operadores Relacionais

Como consta no exemplo anterior, os Operadores Relacionais são amplamente utilizados para realizar validações de dados. A seguir, vamos descrevê-los:

Operador	Descrição
>	Maior que
<	Menor que
>=	Maior ou igual que
<=	Menor ou igual que
==	Igual a
!=	Diferente de
===	Idêntico a
!==	Não idêntico

Note que os operadores `===` e `!==` comparam também o tipo a que pertence o dado. Portanto se compararmos `'5' === 5`, o retorno será **false**, pois mesmo se tratando do mesmo número, ambos pertencem a tipos diferentes (**String** e **Number** respectivamente).

É importante ressaltar que os Operadores Relacionais não possuem ordem de precedência. Portanto, calcula-se aquele que aparece primeiro, da esquerda para a direita.

```
> '5' == 5
true
> '5' === 5
false
> '5' != 5
false
> '5' !== 5
true
> |
```

Figura 16 – Comparação entre `==` e `===`

3.4 Operadores Lógicos

Os Operadores Lógicos tem também como utilidade efetuar comparações entre 2 valores, retornando como resultado valores **booleanos** (**true** ou **false**), porém ao invés de analisar a grandeza de valores como os Operadores Relacionais, analisa apenas se os valores atendem aos requisitos solicitados. Segue exemplo:

```

> var a = 5
undefined
> var b = 8
undefined
> true && false
false
> true && true
true
> false || false
false
> true || false
true
> true || true
true
> a > b && b % 2 == 0
false
> a <= b || b / 2 == 2
true
> 

```

Figura 17 – Execução de Comparações Lógicas

Note que antes das comparações com os Operadores Lógicos serem efetuadas, as operações com os Operadores Relacionais e os Operadores Aritméticos foram executadas, afinal não seria possível fazer uma comparação lógica sem antes obter o valor final do que se deseja comparar.

Portanto, é definida uma ordem de prioridade onde primeiro calcula-se os Operadores Aritméticos, depois os Operadores Relacionais e então os Operadores Lógicos.

Operador	Descrição
!	Negação
&&	Conjunção ou 'E' lógico
	Disjunção ou 'OU' lógico

Ao contrário dos Operadores Relacionais, os Operadores Lógicos possuem ordem de precedência, sendo ela a mesma apresentada na tabela. Primeiro a Negação (!), depois a Conjunção (&&) e então a Disjunção (||).

3.5 Operadores Ternários

Trata-se de um operador que efetua um teste lógico de um dado, da mesma forma que a estrutura de repetição “If Else”, sua sintaxe consiste em:

TESTE LÓGICO ? VERDADEIRO : FALSO

Note que após o operador `?` é definido que deve acontecer caso o resultado do teste lógico seja verdadeiro e após o operador `:` é definido que deve acontecer caso o resultado do teste lógico seja falso.

Veja um exemplo:

```
> var media = 5.5
undefined
> media > 7 ? 'APROVADO' : 'REPROVADO'
'REPROVADO'
> media += 3
8.5
> media
8.5
> media > 7 ? 'APROVADO' : 'REPROVADO'
'APROVADO'
>
```

Figura 18 – Aplicação de Operador Ternário

Perceba que no primeiro teste lógico a `var media` era igual a `5.5` e por isso o retorno foi `'REPROVADO'`, pois `5.5` não é maior que `7`. Quando o valor de `var media` passou a ser `8.5`, o retorno foi `'APROVADO'`, pois `8.5` é que `7`.

Logicamente, conclui-se que o Operador Ternário passa a ser o último na ordem de precedência, pois não é possível efetuar essa validação sem que todos os outros cálculos e validações tenham sido efetuados. Observe outro exemplo, agora com Operações Aritméticas:

```
> var x = 8
undefined
> var res = x % 2 == 0 ? 5 : 9
undefined
> x
8
> res
5
>
```

Figura 19 – Atribuindo valor a uma variável através de validação com Operadores Ternários

Note que nesta validação `5` só deveria ser atribuído como valor de `res` caso o resto da divisão de `8` (valor de `x`) por `2` fosse igual a `0`, e foi o que aconteceu. Caso o valor de `x` fosse igual a `7` por exemplo, o valor atribuído a `res` seria `9`, pois o resto da divisão de `7` por `2` não é igual a `0`.

4 Estruturas Condicionais

As estruturas condicionais são estruturas que geram desvios no código para atender algum caso onde duas ou mais possibilidades são possíveis. Ao analisar a situação a partir das condições definidas, nessas estruturas é definida o que deve ocorrer e então retornar o código ao seu fluxo original. Observe os exemplos:

4.1 Condição Simples

A condição simples analisa somente uma possibilidade de desvio. Exemplo:

```
1  var vel = 60.5
2
3  console.log(`A velocidade do seu carro é ${vel} por hora`)
4
5  if (vel > 60) { //Condição Simples
6    console.log(`Você ultrapassou a velocidade permitida. MULTADO!`)
7  }
8
9  console.log(`Dirija sempre com cinto de segurança`)
```

Figura 20 - Condição Simples

Observe que a condição simples é acionada apenas quando `var vel` é maior do que 60. Portanto, o que está entre parênteses é a condição e o `if (se)` é a estrutura condicional. Lê-se:

“Se `vel` maior que 60, `console.log('Você ultrapassou a velocidade permitida. MULTADO!')`”.

4.2 Condição Composta

A condição composta analisa duas ou mais possibilidades de desvio. Exemplo:

```
1  var país = 'EUA'
2  console.log(`Vivendo em ${país}`)
3
4  if (país == 'Brasil') {
5    console.log('Brasileiro!')
6  } else {
7    console.log('Estrangeiro!')
8  }
```

Figura 21 - Condição Composta

Observe que **else (senão)** não requer uma condição definida, pois essa estrutura já indica o que deve ocorrer caso o **if (se)** não seja verdadeiro. Lê-se:

“Se **país** igual a Brasil, console.log('Brasileiro!'), **senão** console.log('Estrangeiro!')”.

4.3 Switch Case

A estrutura condicional **switch** permite executar um bloco de código diferente de acordo com cada opção (cada **case**) especificada. Seu uso é indicado quando os valores a serem analisados nessas condições são pré-definidos. Observe um exemplo:

```
1  var país = ''
2  console.log(`Vivendo em ${país}`)
3
4  switch (país) {
5      case 'Brasil':
6          console.log('Brasileiro!')
7          break;
8
9      case 'EUA':
10         console.log('Americano!')
11         break;
12
13     case 'Reino Unido':
14     case 'Grã-Bretanha':
15         console.log('Inglês!')
16         break;
17
18     default:
19         console.log('Estrangeiro!');
20 }
```

Figura 22 - Estrutura Condicional Switch

Note que os **cases** estão buscando especificamente o nome dos países através de dados do tipo **String**, é possível utilizar nos **cases** também outros tipos de dados como **Number** e **Boolean**. Além disso, ao não inserir **break** ao **case**, o **case** não é encerrado e se estende até encontrar o próximo **break** ou o fim da estrutura, como pode se observar nos **cases** 'Reino Unido' e 'Grã-Bretanha'.

No fim da estrutura é possível observar também a propriedade **default**, caso nenhum dos dados correspondam aos **cases** anteriores, o valor de **default** é o padrão que será exibido.

5 DOM (Document Object Model)

O Document Object Model (DOM), trata-se do conjunto de objetos presentes no ambiente de um navegador, representando a estrutura de um documento na memória.

Esse modelo se estrutura em formato de árvore, com vários galhos que indicam diferentes elementos da página. Quando esse modelo é alterado através da linguagem de script, se altera a página da web — seja sua estrutura, estilo ou elementos.

Desse modo, o DOM possibilita a manipulação e modificação de elementos de um documento web ao conectar as linguagens de programação à página em questão.

5.1 A Árvore DOM

Partindo do conceito de que o DOM se estrutura em formato de árvore, consideramos como sua raiz o objeto `window` (janela do navegador). Segue fluxograma com outros elementos da árvore:

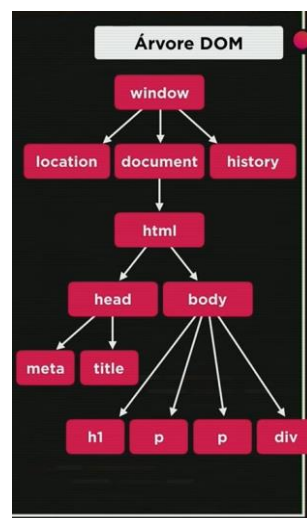


Figura 23 – Árvore DOM

Na Árvore DOM, cada elemento que se encontra abaixo do outro está contido naquele que o precede. Portanto dizemos que aquele elemento é filho daquele que o contém. Por exemplo: o elemento `body` é filho do elemento `html` e o elemento `html` é pai do elemento `body`.

O elemento `location` por sua vez se refere ao elemento do navegador que armazena as informações de URL (Uniform Resource Locator) a página atual e a anterior. Já o elemento `history` que armazena informações como histórico de

navegação e outros. Existem mais elementos a serem considerados na base da Árvore DOM, porém estes são os principais.

5.2 Manipulando o DOM com JavaScript

A importância de conhecer os elementos da Árvore DOM é saber como manipulá-los para fazer diferentes alterações na página. Como já fizemos anteriormente, ao utilizar elementos como:

- `window.alert()`
- `window.prompt()`

Ao utilizar esses comandos. Estamos solicitando por meio de JavaScript que seja emitido um alerta e um prompt para entrada de dados através da janela do navegador respectivamente.

Utilizamos anteriormente o comando:

- `document.write()`

Através dele é possível escrever informações diretamente no documento da página que, como vimos previamente no fluxograma, contém toda a estrutura que comumente estamos desenvolvendo.

Para acessar os elementos presentes na Árvore DOM, podemos selecioná-los através de 5 métodos diferentes. Sendo eles:

- Por Marca (`getElementsByTagName()`);

```
<body>
  <h1>Iniciando estudos DOM</h1>
  <p>Aqui vai o resultado</p>
  <p>Aprendendo a usar o <strong>DOM</strong> em JavaScript</p>
  <div>Clique em mim</div>
  <script>
    var p1 = window.document.getElementsByTagName('p')[0]
    window.document.write('Está escrito assim: ' + p1.innerText)
  </script>
</body>
```

Figura 24 – Método de Seleção por Marca

Note que o método `getElementsByTagName()` está buscando a tag `<p>` em sua seleção e logo após vemos o índice `[0]`, que tem como função indicar que queremos selecionar a tag `<p>` de índice `[0]` na página (correspondente a `<p>Aqui vai o resultado</p>`).

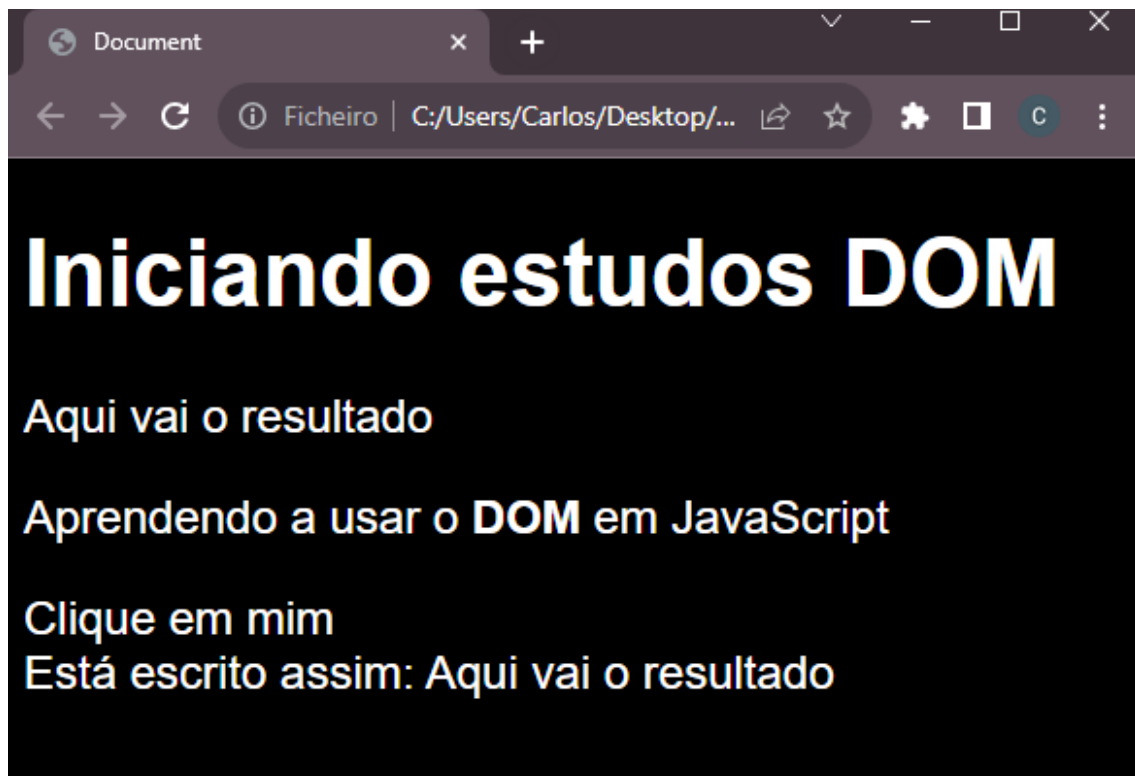


Figura 25 – Retorno do Método de Seleção por Marca

- Por ID (`getElementById()`);

```
<body>
  <h1>Iniciando estudos DOM</h1>
  <p>Aqui vai o resultado</p>
  <p>Aprendendo a usar o <strong>DOM</strong> em JavaScript</p>
  <div id="msg">Clique em mim</div>
  <script>
    var d = document.getElementById('msg');
    d.style.color = "#FF00";
    d.innerText = "Estou aguardando..."
  </script>
</body>
```

Figura 26 – Método de Seleção por ID

Iniciando estudos DOM

Aqui vai o resultado

Aprendendo a usar o **DOM** em JavaScript

Estou aguardando...

Figura 27 – Retorno do Método de Seleção por ID

- Por Nome (`getElementsByName()`);

```
<body>
  <h1>Iniciando estudos DOM</h1>
  <p>Aqui vai o resultado</p>
  <p>Aprendendo a usar o <strong>DOM</strong> em JavaScript</p>
  <div name="msg">Clique em mim</div>
  <script>
    var d = document.getElementsByName('msg')[0];
    d.style.color = "#FF0";
    d.innerText = "Estou aguardando..."
  </script>
</body>
```

Figura 28 – Método de Seleção por Nome

Note que este método espera que o mesmo atributo `name` seja utilizado em outros elementos (pois `Elements` está no plural) portando é necessário também adicionar um índice ao utilizar este método. Seu retorno é logicamente o mesmo do método `getElementById()`.

- Por Classe (`getElementsByClassName()`);

Funciona de forma muito semelhante ao método `getElementsByName()`, porém utiliza o atributo `class` ao invés do atributo `name`. Veja o exemplo:

```

<body>
  <h1>Iniciando estudos DOM</h1>
  <p>Aqui vai o resultado</p>
  <p>Aprendendo a usar o <strong>DOM</strong> em JavaScript</p>
  <div class="msg">Clique em mim</div>
  <script>
    var d = document.getElementsByClassName('msg')[0];
    d.style.color = "#FF0";
    d.innerText = "Estou aguardando..."
  </script>
</body>

```

Figura 29 – Método de Seleção por Classe

Seu retorno logicamente é o mesmo do método `getElementsByName()`.

- Por Seletor (`querySelector()` e `querySelectorAll()`);

Os métodos `querySelector()` e `querySelectorAll()` por sua vez possuem aplicação mais ampla por terem sido inseridos em uma versão mais recente do ECMAScript.

O método `querySelector()` tem como princípio retornar o primeiro elemento encontrado na busca. Segue exemplo:

```

1 <!DOCTYPE html>
2 <html lang="pt-br">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1">
6   <title>Document</title>
7 </head>
8 <body>
9   <h1>Iniciando estudos DOM</h1>
10  <p class="msg">Aqui vai o resultado</p>
11  <p>Aprendendo a usar o <strong>DOM</strong> em JavaScript</p>
12  <div class="msg">Clique em mim</div>
13  <script>
14    var d = document.querySelector('.msg');
15    d.style.color = "#FF0";
16    d.innerText = "Estou aguardando..."
17  </script>
18 </body>
19 </html>

```

Iniciando estudos DOM

Estou aguardando...

Aprendendo a usar o **DOM** em JavaScript

Clique em mim

Figura 30 – Aplicação e Retorno de `querySelector()`

Note que mesmo havendo dois elementos com o atributo `class="msg"`, apenas o primeiro elemento encontrado na busca sofreu as alterações orientadas.

Já o método `querySelectorAll()` tem como objetivo selecionar todos os elementos encontrados na busca e então indicar a próxima ação. Observe:

```
<body>
  <h1>Iniciando estudos DOM</h1>
  <p class="msg">Aqui vai o resultado</p>
  <p>Aprendendo a usar o <strong>DOM</strong> em JavaScript</p>
  <div class="msg">Clique em mim</div>
  <script>
    var d = document.querySelectorAll('.msg');
    console.log(d);
  </script>
</body>
```

Figura 31 – Aplicação de querySelectorAll()

Note que foi utilizado `console.log()` neste exemplo, este é um comando utilizado para exibir dados no console do navegador e/ou no console do editor de código.

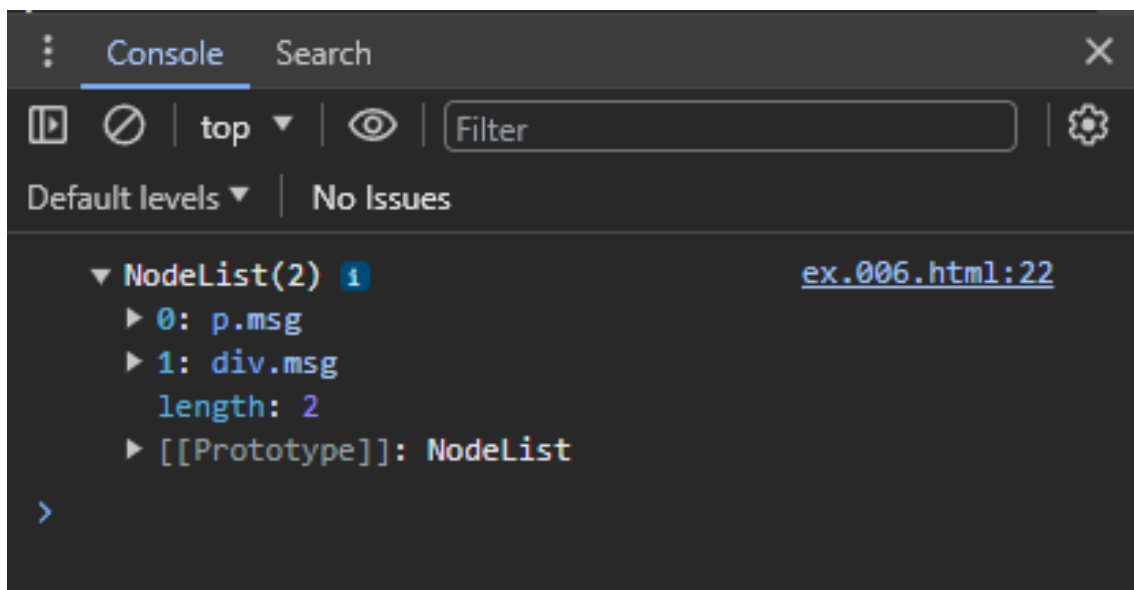


Figura 32 – Retorno da Aplicação de querySelectorAll()

O método `querySelectorAll()` é comumente utilizado para selecionar determinada quantidade de elementos a partir de um atributo em comum e então indicar um comportamento que estes elementos terão após a utilização de algum novo comando.

5.2.1 Alterações de estilo por meio de JavaScript

Uma das alterações no DOM que é possível realizar através de JavaScript são os estilos CSS e agora mostraremos como:

```
<body>
  <h1>Iniciando estudos DOM</h1>
  <p>Aqui vai o resultado</p>
  <p>Aprendendo a usar o <strong>DOM</strong> em JavaScript</p>
  <div>Clique em mim</div>
  <script>
    var corpo = window.document.body;

    corpo.style.background = "blue";
  </script>
```

Figura 33 – Alteração de background com JavaScript

Note que `var corpo` foi declarada somente para representar o caminho padrão para alcançar o `body`, em sequência adicionamos a propriedade `style` e então a propriedade `background` que tem como valor “**blue**”, gerando o seguinte resultado:

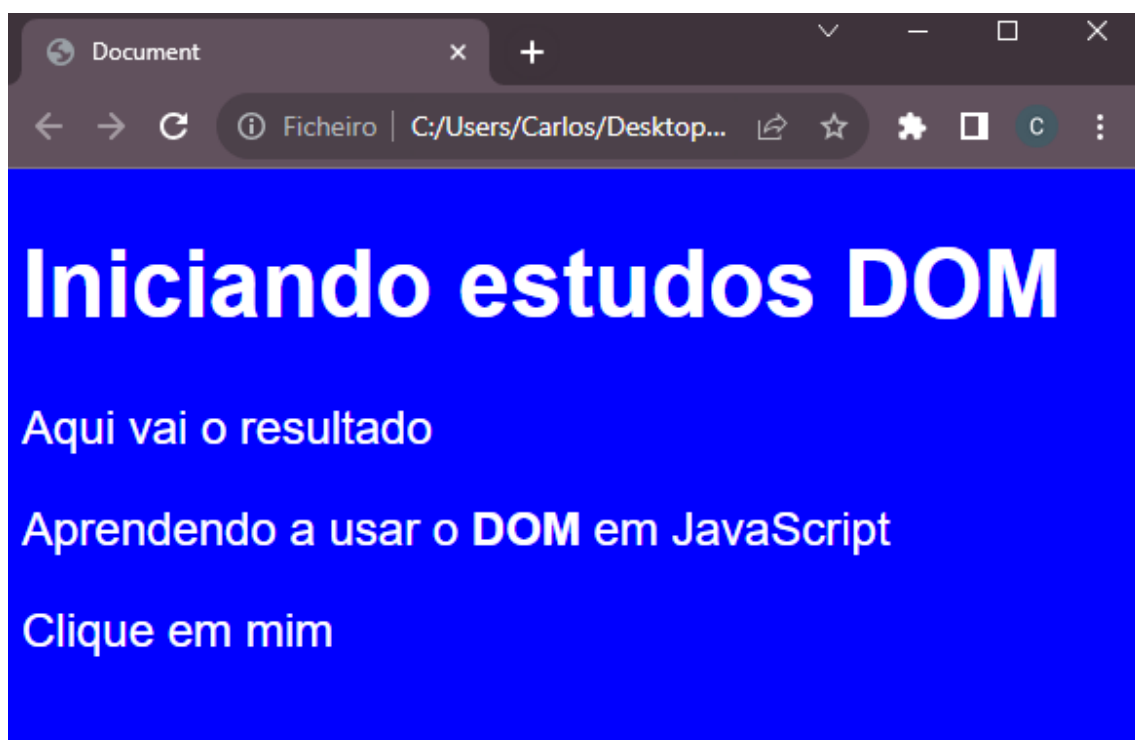


Figura 34 – Retorno da Alteração de background com JavaScript

```

<!DOCTYPE html>
<html lang="pt-br">
  <head> ... </head>
  ... <body style="background: blue;" == $0
    <h1>Iniciando estudos DOM</h1>
    <p>Aqui vai o resultado</p>
    <p>...</p>
    <div>Clique em mim</div>

```

Figura 35 – Produto da Alteração de background com JavaScript

Caso trate-se de uma alteração cujo nome possua mais que 2 palavras, utilizamos Camel Case:

```

<body>
  <h1>Iniciando estudos DOM</h1>
  <p>Aqui vai o resultado</p>
  <p>Aprendendo a usar o <strong>DOM</strong> em JavaScript</p>
  <div>Clique em mim</div>
  <script>
    var corpo = window.document.body;

    corpo.style.width = "100%";
    corpo.style.display = "flex";
    corpo.style.flexDirection = "column";
    corpo.style.alignItems = "center";
  </script>
</body>

```

Figura 36 – Alterações com mais de 2 palavras



Figura 37 – Retorno dessas Alterações com mais de 2 palavras


```

<!DOCTYPE html>
<html lang="pt-br">
  <head>...</head>
  <body style="width: 100%; display: flex; flex-direction:
    column; align-items: center;"> flex == $0
    <h1>Iniciando estudos DOM</h1>
    <p>Aqui vai o resultado</p>
    <p>...</p>
    <div>Clique em mim</div>

```

Figura 38 – Produto dessas Alterações com mais de 2 palavras

5.2.2 .innerText e .innerHTML

A seguir, confira que o método utilizado para exibir a tag <p> de índice [1] na página foi `.innerText` (que insere um texto sem formatação pré-definida, ignorando inclusive tags como).

```

<body>
  <h1>Iniciando estudos DOM</h1>
  <p>Aqui vai o resultado</p>
  <p>Aprendendo a usar o <strong>DOM</strong> em JavaScript</p>
  <div>Clique em mim</div>
  <script>
    var p1 = window.document.getElementsByTagName('p')[1]
    window.alert(p1.innerText)
  </script>
</body>

```

Figura 39 - Utilização de `.innerText`

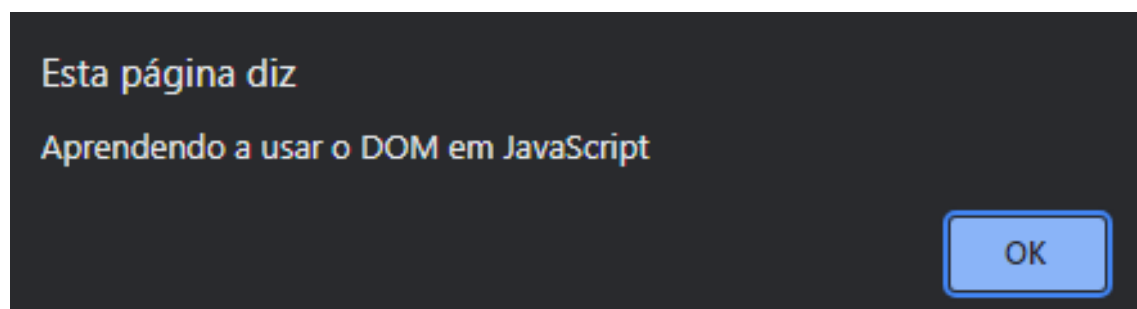


Figura 40 - Retorno de Utilização de `.innerText`

É possível utilizar também `.innerHTML` (que insere um texto com a formatação do elemento original, inclusive a própria tag ``)

```
<body>
  <h1>Iniciando estudos DOM</h1>
  <p>Aqui vai o resultado</p>
  <p>Aprendendo a usar o <strong>DOM</strong> em JavaScript</p>
  <div>Clique em mim</div>
  <script>
    var p1 = window.document.getElementsByTagName('p')[1]
    window.alert(p1.innerHTML)
  </script>
</body>
```

Figura 41 – Utilização de `.innerHTML`

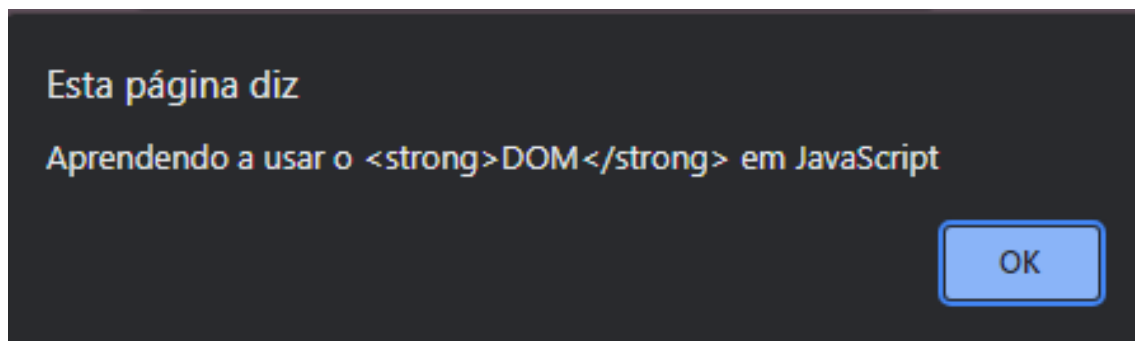


Figura 42 – Retorno de Utilização de `.innerHTML`

Observe a diferença de ambos com o `document.write()`:

```
<body>
  <h1>Iniciando estudos DOM</h1>
  <p>Aqui vai o resultado</p>
  <p>Aprendendo a usar o <strong>DOM</strong> em JavaScript</p>
  <div>Clique em mim</div>
  <script>
    var p1 = window.document.getElementsByTagName('p')[1]
    document.write(p1.innerText)
  </script>
</body>
```

Figura 43 – Utilização de `.innerText` em `document.write()`

Iniciando estudos DOM

Aqui vai o resultado

Aprendendo a usar o **DOM** em JavaScript

Clique em mim

Aprendendo a usar o **DOM** em JavaScript

Figura 44 – Retorno de Utilização de .innerText em document.write()

Note que o texto foi inserido sem o negrito, ou seja, a tag foi desconsiderada. Agora veja com .innerHTML:

```
<body>
  <h1>Iniciando estudos DOM</h1>
  <p>Aqui vai o resultado</p>
  <p>Aprendendo a usar o <strong>DOM</strong> em JavaScript</p>
  <div>Clique em mim</div>
  <script>
    var p1 = window.document.getElementsByTagName('p')[1]
    document.write(p1.innerHTML)
  </script>
</body>
```

Figura 45 – Utilização de .innerHTML em document.write()

Iniciando estudos DOM

Aqui vai o resultado

Aprendendo a usar o **DOM** em JavaScript

Clique em mim

Aprendendo a usar o **DOM** em JavaScript

Figura 46 – Retorno de Utilização de `.innerHTML` em `document.write()`

Com `.innerHTML` é possível inserir também tags HTML ao DOM.

5.3 Eventos DOM

Os eventos DOM são os eventos que podem acontecer em determinada situação. Citaremos como exemplo os eventos de mouse:

- `click` – Ativado ao clicar no elemento;
- `mouseenter` – Ativado quando o mouse entra no elemento;
- `mousemove` – Ativado quando o mouse se move dentro do elemento;
- `mouseout` – Ativado quando o mouse sai do elemento;
- `mousedown` – Ativado quando o mouse está pressionado dentro do elemento;
- `mouseup` – Ativado ao soltar o mouse que estava pressionado dentro do elemento;

5.3.1 Funções

Funções são blocos de código que devem ser executados apenas quando solicitado, quando houver algum evento específico que irá iniciar a execução da função. Exemplo:

```

<body>
  <div id="area" onclick="clicar()">
    Interaja...
  </div>

  <script>
    function clicar() {
      var a = window.document.getElementById('area')
      a.innerText = 'Clicou!'
    }
  </script>
</body>

```

Figura 47 - Aplicação de Função

Note que o objetivo dessa função é alterar o texto presente no elemento que possui **id="area"**, o método escolhido para fazer essa alteração foi o **onclick**, ou seja, ao clicar no elemento em questão, a função **clicar()** foi executada.

Observe também como é possível adicionar diferentes funções ao mesmo elemento:

```

<body>
  <div id="area" onclick="clicar()" onmouseenter="entrar()" onmouseout="sair()">
    Interaja...
  </div>

  <script>
    var a = window.document.getElementById('area')

    function clicar() {
      a.innerText = 'Clicou!'
      a.style.cursor = 'auto'
    }
    function entrar() {
      a.innerText = 'Entrou!'
      a.style.cursor = 'pointer'
    }
    function sair() {
      a.innerText = 'Saiu!'
    }
  </script>
</body>

```

Figura 48 - Aplicação de várias funções ao mesmo elemento

Neste exemplo foi aplicado também uma alteração de estilo no cursor do mouse, para tornar mais intuitivo a navegação na página.

Agora apresentaremos uma alternativa para este método, o Event Listener.

5.3.1.1 Event Listener

O Event Listener funciona como um ouvidor de eventos no JavaScript, sua função é captar ações que estão ocorrendo na página e executar funções conforme lhe for programado. Observe:

```
<body>
  <div id="area">
    Interaja...
  </div>

  <script>
    var a = window.document.getElementById('area')
    a.addEventListener('click', clicar)
    a.addEventListener('mouseenter', entrar)
    a.addEventListener('mouseout', sair)

    function clicar() {
      a.innerText = 'Clicou!'
      a.style.cursor = 'auto'
    }
    function entrar() {
      a.innerText = 'Entrou!'
      a.style.cursor = 'pointer'
    }
    function sair() {
      a.innerText = 'Saiu!'
    }
  </script>
</body>
```

Figura 49 - Utilização de Event Listener

Note que a utilização do método `addEventListener()` gera um HTML mais limpo, pois anula a necessidade de adicionar atributos à tag que pretendemos alterar.

6 Estruturas de Repetição

As Estruturas de Repetição permitem executar mais de uma vez um mesmo trecho de código. Trata-se de uma forma de executar blocos de comandos somente sob determinadas condições, mas com a opção de repetir o mesmo bloco quantas vezes for necessário.

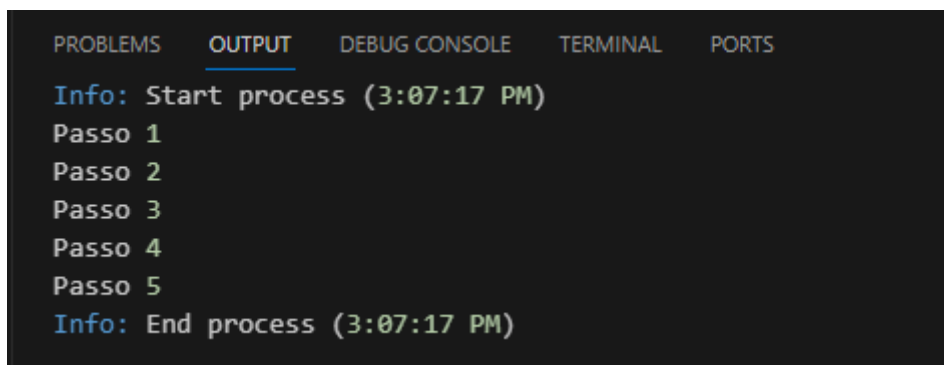
6.1 While

A palavra “while” significa “enquanto” em português, portanto, lê-se: – “Enquanto a expressão booleana for verdadeira, execute os comandos do bloco abaixo”. – Ou seja, o bloco de comandos será repetido enquanto a expressão booleana for verdadeira. Observe um exemplo prático:

```
1  var contador = 1
2  while (contador < 6) {
3      console.log(`Passo ${contador}`)
4      contador++
5  }
```

Figura 50 - Estrutura de Repetição While

Note que a `var contador` tem como princípio proporcionar a contagem de 1 à 6, portanto é necessário definir no `while` a forma como essa contagem será efetuada para que o loop não se estenda infinitamente, a forma escolhida foi `contador++` (`contador = contador + 1`). Seu retorno portanto é:



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
Info: Start process (3:07:17 PM)
Passo 1
Passo 2
Passo 3
Passo 4
Passo 5
Info: End process (3:07:17 PM)
```

Figura 51 - Retorno da Estrutura de Repetição While

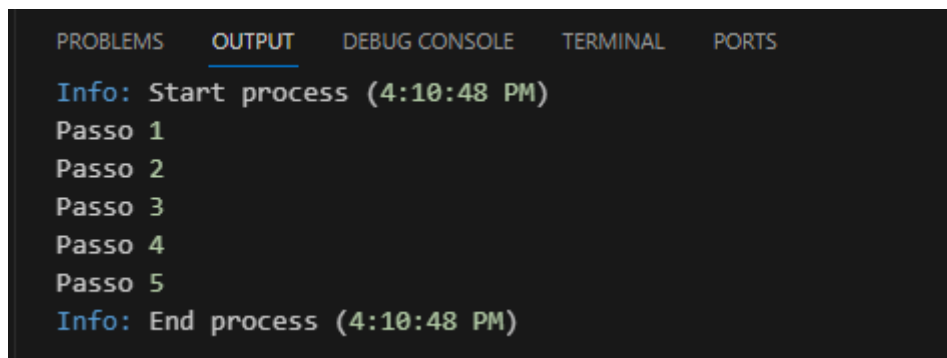
6.2 Do While

A diferença entre o `while` para o `do while` é que, no `do while` sempre acontece a primeira execução do bloco de comandos e a expressão booleana só é avaliada ao final de cada execução. Observe um exemplo prático:

```
13  do {  
14      console.log(`Passo ${contador}`)  
15      contador++  
16  } while (contador < 6)
```

Figura 52 - Estrutura de Repetição Do While

Observe agora o retorno desse loop:



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  
Info: Start process (4:10:48 PM)  
Passo 1  
Passo 2  
Passo 3  
Passo 4  
Passo 5  
Info: End process (4:10:48 PM)
```

Figura 53 - Retorno da Estrutura de Repetição Do While

Neste simples algoritmo utilizado para exemplificação, nota-se que tanto o `while` quanto o `do while` apresentam retornos semelhantes, então em qual situação optar por `while` ou `do while`?

6.3 Quando optar por While ou Do While?

A diferença fundamental entre o `while` e `do while` é que o `while` verifica a condição antes de executar o bloco de código, enquanto o `do while` executa o bloco de código pelo menos uma vez e, em seguida, verifica a condição.

Portanto, no `while` se a condição for falsa desde o início, o bloco de código nunca será executado. Já no `do while`, o código será executado ao menos uma vez.

6.4 For

O **for** é uma estrutura de repetição na qual seu ciclo será executado por um tempo ou condição pré-determinados e em uma quantidade de vezes que determinamos. Quando utilizamos o **for**, precisamos de uma variável para auxiliar a controlar a quantidade de repetições a serem executadas. Observe um exemplo:

```
1  var contador = 1
2
3  // Estrutura de Repetição For
4
5  for (contador = 1; contador <= 10; contador++) {
6      console.log(`Passo: ${contador}`)
7  }
```

Figura 54 - Estrutura de Repetição For

Observe agora o retorno dessa estrutura de repetição:

```
Info: Start process (2:26:22 PM)
Passo: 1
Passo: 2
Passo: 3
Passo: 4
Passo: 5
Passo: 6
Passo: 7
Passo: 8
Passo: 9
Passo: 10
Info: End process (2:26:22 PM)
```

Figura 55 - Retorno da Estrutura de Repetição For

7 Depuração de Código

A seguir observe uma funcionalidade presente no editor de código VS Code para depuração de código:

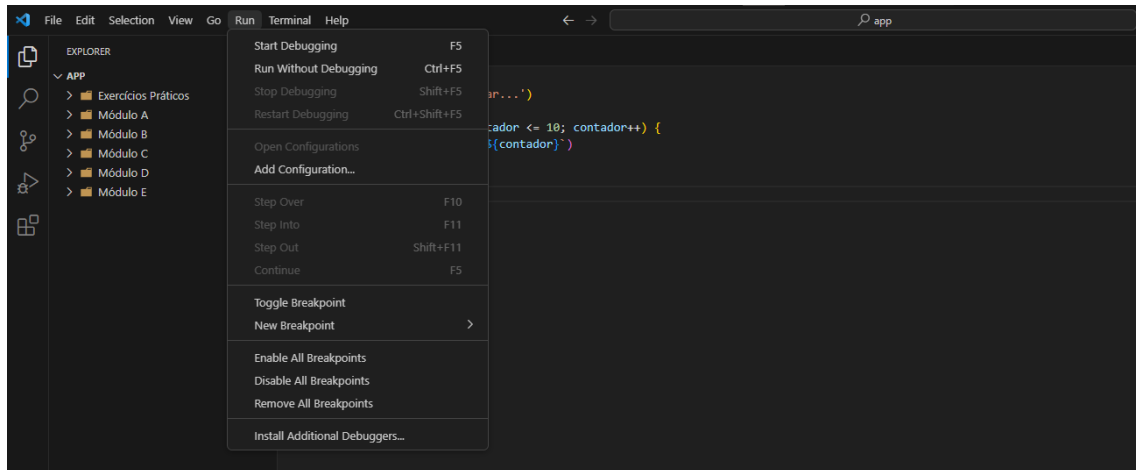


Figura 56 - Abrindo a Ferramenta de Debug

Selecione a opção “Start Debugging” ou pressione a tecla de atalho F5.

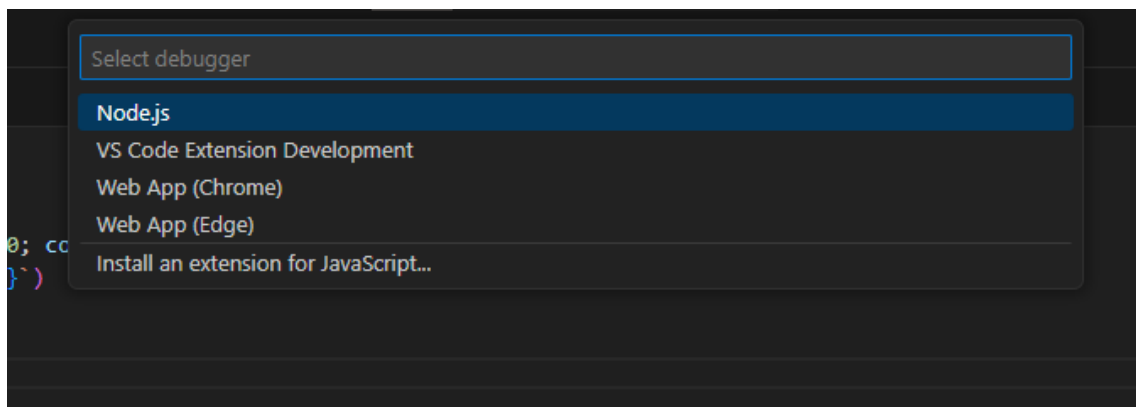


Figura 57 - Seleção da Ferramenta de Depuração

Nesta ocasião, selecionaremos a ferramenta “Node.js”.

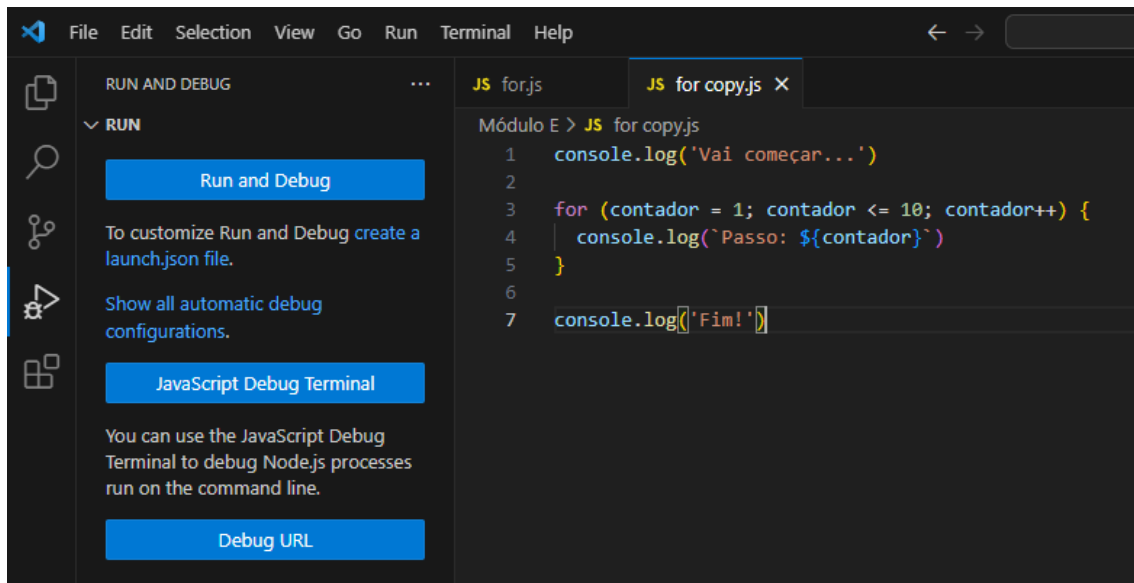


Figura 58 - Seleção de opções de depuração

Selecione “create a launch.json file” e em seguida selecione a opção sugerida “Node.js”.

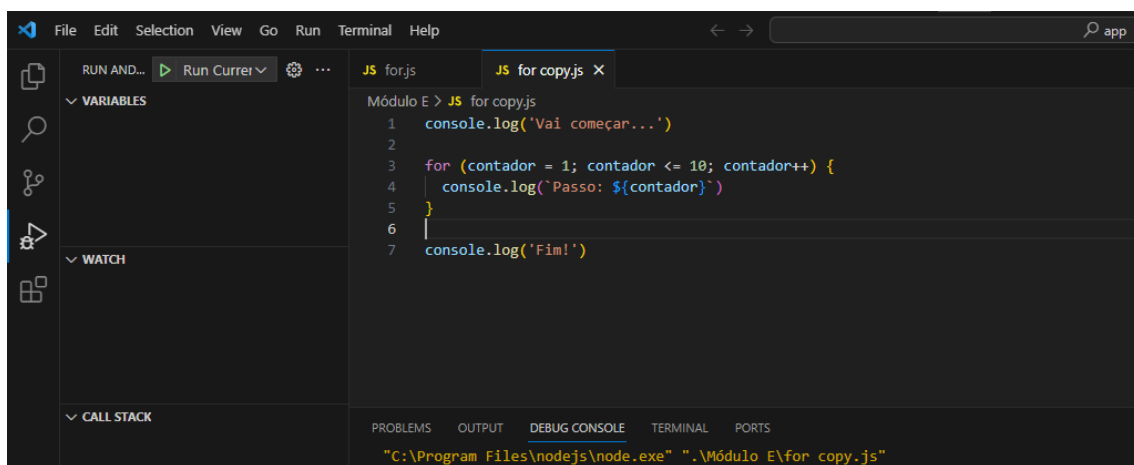


Figura 59 - Abertura do Menu Lateral e Console de Depuração

Note que agora está aberto um Menu Lateral com opções de depuração e também a opção “Debug Console”.

Ao lado dos marcadores de linhas no editor, é possível selecionar os chamados **breakpoints**, que são pontos de parada na depuração que irão nos auxiliar na análise trecho a trecho do código:

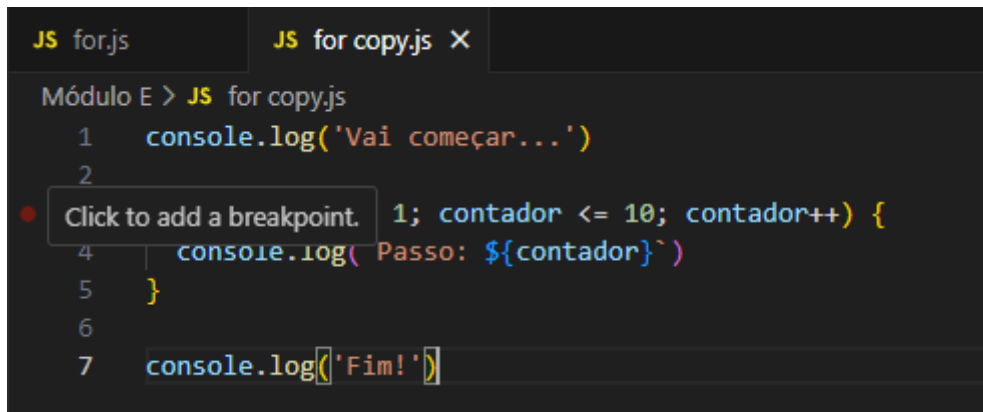


Figura 60 - Opção de Adição de Breakpoints

Clicando em determinado **Breakpoint**, selecionamos o trecho de código que será depurado, nessa ocasião selecionaremos as linhas **1** e **7**.

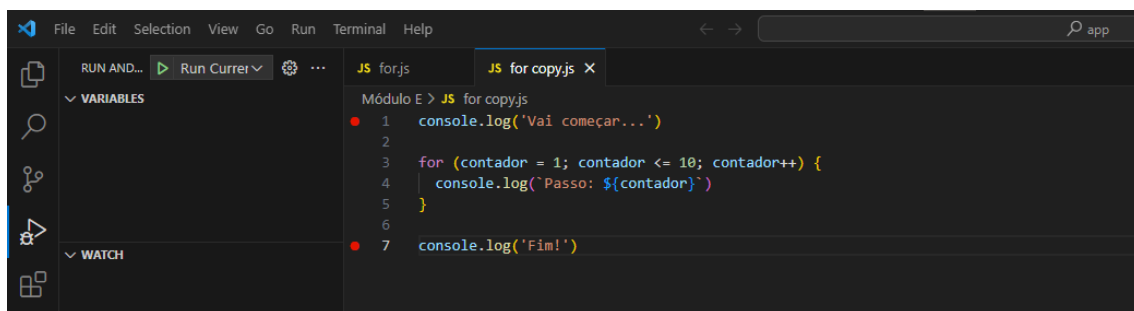


Figura 61 - Seleção do trecho de código a ser depurado

A fim de monitorar o comportamento da **var contador**, a adicionaremos ao campo **Watch**:

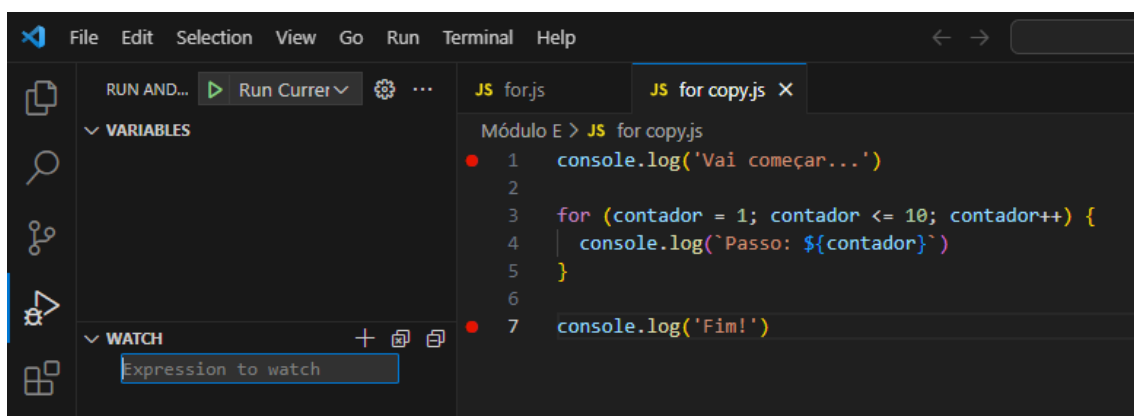


Figura 62 - Adição de variável ao campo Watch

No campo “Expression to watch”, adicionaremos a **var contador**:

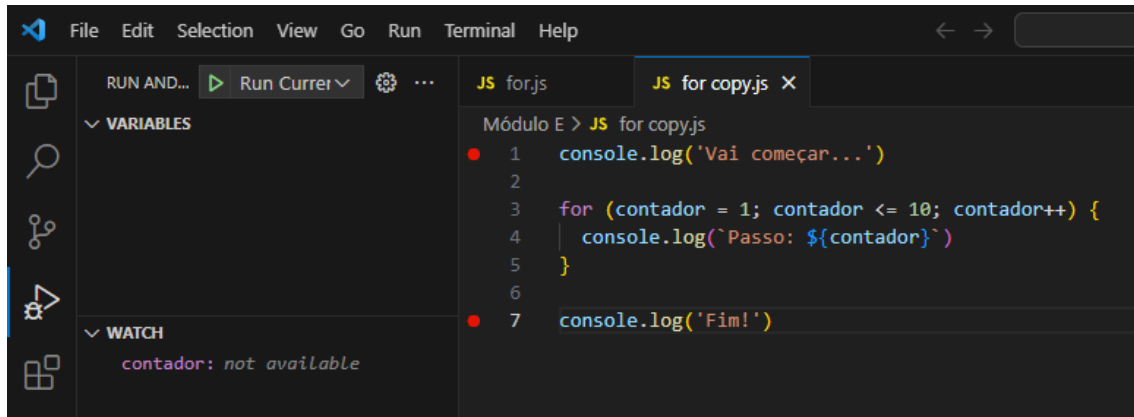


Figura 63 - Variável adicionada no campo Watch

Note que no momento a variável conta com a instrução “not available” atribuída a ela. Isso ocorre porque o trecho de código não está sendo executado no momento. Portanto, clicaremos na opção de Play na área superior do menu lateral.

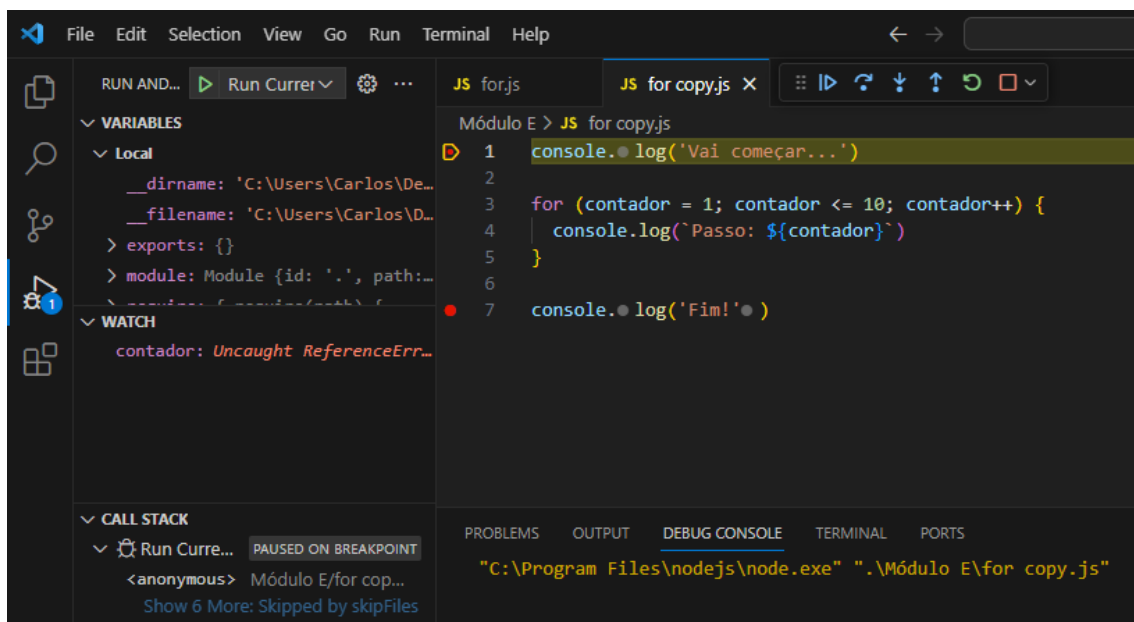


Figura 64 - Código em andamento

Note que quando o código está em andamento, um menu de controle é disponibilizado. Suas opções são:

- 1ª opção: Permite mover a opção do Menu de Controle;
- 2ª opção: **Continue (F5)**, Pausa a execução do código;
- 3ª opção: **Step Over (F10)**: Avança para os próximos passos do código;
- 4ª opção: **Step Into (F11)**: Avança para a próxima linha, porém se a próxima linha for a execução de um método, ele irá para a primeira linha dentro desse método;
- 5ª opção: **Step Out (Shift + F11)**: Avança para a próxima linha, saindo do método corrente;
- 6ª opção: **Restart (Ctrl + Shift + F5)**: Reinicia a execução do código;
- 7ª opção: **Stop (Shift + F5)**: Encerra a execução do código;

Como pretendemos depurar passo a passo do código, utilizaremos a opção **Step Over (F10)**.

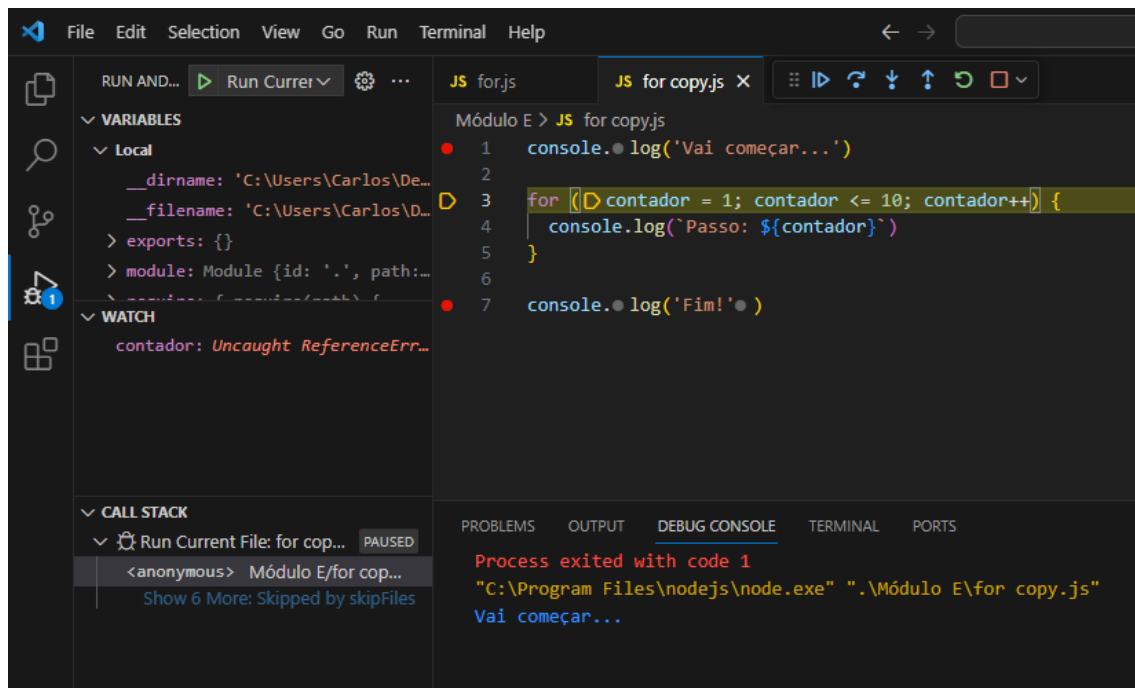


Figura 65 - Utilização da opção Step Over (F10)

Note que o primeiro **console.log()** foi executado e seu retorno está sendo exibido no **Debug Console**, atualmente o código está depurando a primeira condição do **for**, ao pressionar **Step Over (F10)** novamente, as próximas condições do **for** serão executadas, gerando o seguinte retorno:

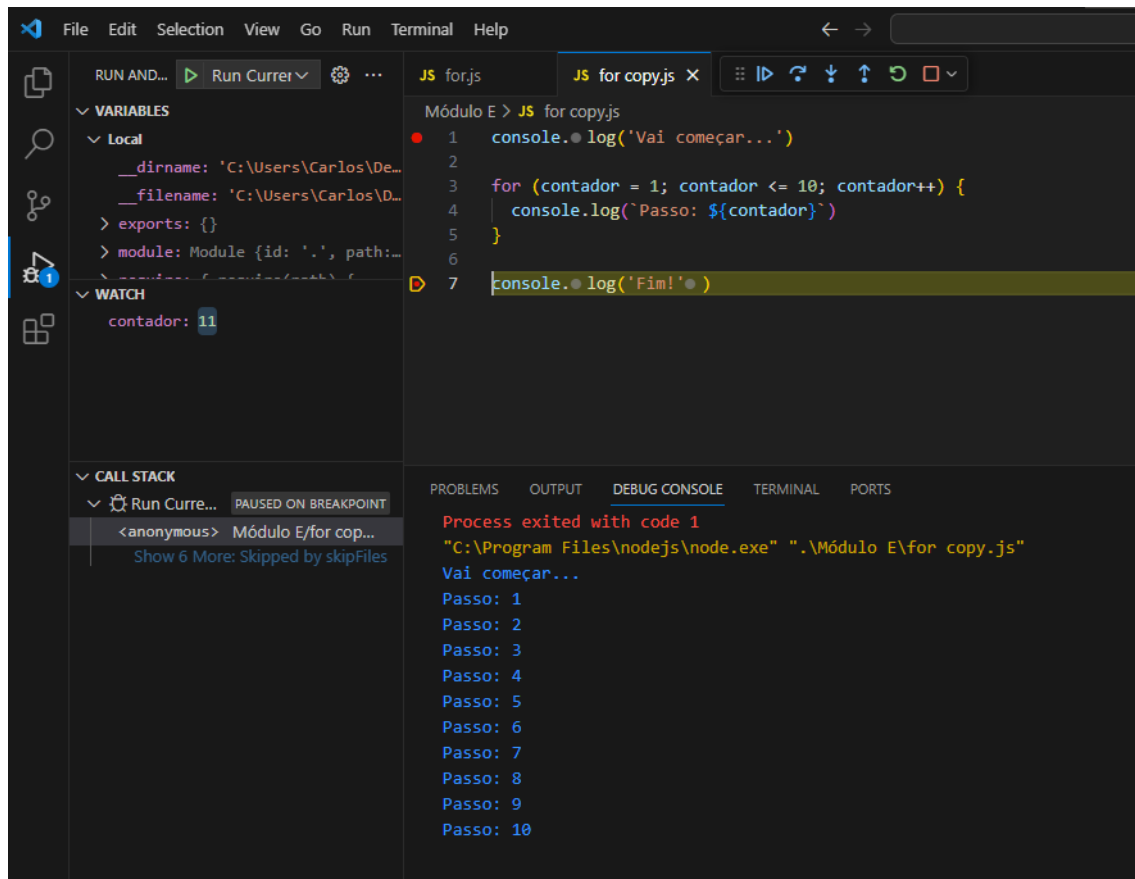


Figura 66 - Passo seguinte na depuração do código

Note que em **Watch** o valor da **var contador** é **11**, porém o valor indicado no **Debug Console** é **10**. Isso ocorre pois o **for** é uma estrutura de repetição que faz verificações a partir das condições previamente estipuladas (início igual à 1, fim menor ou igual à 10 e contagem de 1 em 1), a última verificação realizada (**contador: 11**), retornou **False**, encerrando então as verificações.

8 Variáveis Compostas (Arrays (ou Vetores))

As Variáveis Compostas são classificadas em **homogêneas** (vetoriais) ou **heterogêneas** (registros). Definindo:

- **Homogêneas**: compostas por dados de um mesmo tipo;
- **Heterogêneas**: compostas por diferentes tipos de dados.

Em função de sua capacidade de armazenar diferentes valores, as Variáveis Compostas podem ser encaradas como “estruturas” de armazenamento.

O **Array** (nome comumente utilizado para se referir à Variáveis Compostas) é portanto uma variável especial, utilizada para armazenar uma grande quantidade de valores em um só lugar ou nome. Os valores podem ser acessados por meio de um número de índice.

Observe um exemplo de Variável Composta Homogênea:

```
1 let numbers = [1, 2, 3]
2
3 console.log(`Os valores do Array numbers são ${numbers}`)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

"C:\Program Files\nodejs\node.exe" ".\Módulo F\ambiente.js"
Os valores do Array numbers são 1,2,3

Figura 67 - Variável Composta Homogênea

Observe agora um exemplo de Variável Composta Heterogênea:

```
1 let numbers = [1, 'String', true]
2
3 console.log(`Os valores do Array numbers são ${numbers}`)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

"C:\Program Files\nodejs\node.exe" ".\Módulo F\ambiente.js"
Os valores do Array numbers são 1, String,true

Figura 68 - Variável Composta Heterogênea

Estes são simples exemplos que buscam mostrar a variedade de possibilidades ao se armazenar dados em um **Array**. Observe a seguir diferentes formas de se manipular **Arrays**.

8.1 Exibição de elementos de Arrays através do for

Uma prática comum relacionada aos **Arrays** é exibir seus valores em determinadas ocasiões, para isso, utiliza-se a estrutura de repetição **For**. Observe um exemplo:

```
1  let numbers = [1, 9, 5, 3, 8]
2
3  for (let pos=0; pos<numbers.length; pos++) {
4      console.log(numbers[pos])
5  }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

"C:\Program Files\nodejs\node.exe" ".\Módulo F\ambiente.js"

1
9
5
3
8

Figura 69 - Exibição de Elementos de um Array através do For

Porém, o **For** possui uma sintaxe única para ser utilizada nessa ocasião, sua lógica consiste em:

Para cada elemento (x) em Array (y)

Observe um exemplo:

```
1 let numbers = [1, 9, 5, 3, 8]
2
3 for (let pos in numbers) {
4   console.log(numbers[pos])
5 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

"C:\Program Files\nodejs\node.exe" ".\Módulo F\ambiente.js"

1
9
5
3
8

Figura 70 - Utilização de sintaxe especial do For

Concluindo, trata-se de uma forma simplificada de exibir esses elementos.

8.2 Manipulação de Arrays

Neste capítulo iremos apresentar diferentes formas de se manipular os dados presentes em Arrays:

- `n[índice] = valor` – Ao indicar o índice, é possível selecionar em que posição do Array deseja que o valor seja armazenado;
- `n.push(valor)` – Insere o valor diretamente na última posição vazia do Array;
- `n.length` – Exibe o comprimento do Array, a quantidade de elementos presentes.
- `n.sort()` – Considerando um Array com dados do tipo Number, reorganiza os elementos e os exibe em ordem crescente;
- `n.indexOf(valor)` – Exibe o índice do Array onde se localiza o valor especificado. Caso o valor não seja encontrado no Array, por padrão o valor **-1** é retornado.

9 Objetos

A linguagem JavaScript é projetada com base em um simples paradigma orientado a objeto. Um objeto é uma coleção de propriedades, e uma propriedade é uma associação entre um nome (ou chave) e um valor. Um valor de propriedade pode ser uma função, que é então considerada um método do objeto. Observe um exemplo:

```
1  let amigo = {
2      nome: 'José',
3      sexo: 'M',
4      peso: 85.4,
5      engordar(p) {
6
7      }
8  }
9
10 console.log(amigo)
11 console.log(typeof(amigo))
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

"C:\Program Files\nodejs\node.exe" ".\Módulo F\ex.013\objeto01.js"

> {nome: 'José', sexo: 'M', peso: 85.4, engordar: f}

object

Figura 71 - Exemplo de Objeto

Note que, conforme indica o exemplo, os objetos recebem propriedades que possuem um dado de determinado tipo, assim como a propriedade **nome** possui o dado do tipo **String** 'José'. Além disso, é possível também declarar funções em objetos. Observe agora como acessar dados específicos de um objeto:

```
1  let amigo = {
2    nome: 'José',
3    sexo: 'M',
4    peso: 85.4,
5    engordar(p) {
6
7    }
8  }
9
10 console.log(amigo.nome)
11 console.log(typeof(amigo))
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

"C:\Program Files\nodejs\node.exe" ".\Módulo F\ex.013\objeto01.js"

José
object

Figura 72 - Acessando Dado de um objeto a partir de sua Propriedade

Note que ao indicar a propriedade, apenas o dado desejado foi exibido. Agora observe a utilização da função presente nesse objeto:

```
1  let amigo = {
2    nome: 'José',
3    sexo: 'M',
4    peso: 85.4,
5    engordar(p=0) {
6      console.log('Engordou')
7      this.peso += p
8    }
9  }
10
11 amigo.engordar(2)
12 console.log(`${amigo.nome} pesa ${amigo.peso}Kg`)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

"C:\Program Files\nodejs\node.exe" ".\Módulo F\ex.013\objeto01.js"

Engordou
José pesa 87.4Kg

Figura 73 - Alterando Dados em funções presentes em Objetos

10 Funções

Uma Função é um bloco de código que executa alguma operação. Opcionalmente, uma função pode definir parâmetros de entrada que permitem que os chamadores passem argumentos para a função. Uma função também pode retornar um valor como saída. Em resumo, são ações executadas assim que são chamadas ou em decorrência de algum evento. Observe um exemplo:

```
1  function parimpar(n) {
2      if (n % 2 == 0) {
3          return 'Par!'
4      } else {
5          return 'Ímpar!'
6      }
7  }
8
9  let res = parimpar(4)
10 console.log(res)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

"C:\Program Files\nodejs\node.exe" ".\Módulo F\ex.012\funcao01.js"

Par!

Figura 74 - Exemplo de Função

Note que em **let res**, o valor definido foi a função **parimpar(4)**, sendo **4** o parâmetro definido para a função, com a validação executada, concluiu-se que **4** é par, portanto, o valor atribuído à **let res** passou a ser **'Par!'**.

Observe agora uma função que possui 2 parâmetros:

```
1  function soma(n1 = 0, n2 = 0) {
2      return n1 + n2
3  }
4
5  console.log(soma(2, 5))
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

"C:\Program Files\nodejs\node.exe" ".\Módulo F\ex.012\funcao02.js"

7

Figura 75 - Exemplo de Função com 2 ou mais parâmetros

Note que para `var n1` e `var n2` o valor inicial 0 foi atribuído, essa escolha ocorreu devido a possibilidade de um dos parâmetros não ser passado para a função, e assim não gerando o resultado desejado. Observe:

```
1 function soma(n1, n2) {
2   return n1 + n2
3 }
4
5 console.log(soma(2))
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

"C:\Program Files\nodejs\node.exe" ".\Módulo F\ex.012\funcao02.js"

NaN

Figura 76 - Exemplo de retorno não esperado por falta de parâmetros

O valor **NaN (Not a Number)** foi retornado pois não foi atribuído um valor à `var n2` e portanto não haviam dois valores numéricos para efetuar a soma.

Agora observe um exemplo onde definimos uma função como valor de uma variável:

```
1 let v = function(x) {
2   return x*2
3 }
4
5 console.log(v(5))
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

"C:\Program Files\nodejs\node.exe" ".\Módulo F\ex.012\funcao03.js"

10

Figura 77 - Exemplo de função definida como valor de variável

Note que através dessa técnica foi possível atribuir um parâmetro a uma variável, afinal, o tipo dessa variável passou a ser **Function**.

10.1 Hoisting

A título de curiosidade, a linguagem de programação JavaScript possui um comportamento conhecido como **Hoisting**, que consiste em içar as declarações de variável para o topo antes da execução do código, portanto, o primeiro código exibido no capítulo anterior foi executado com a seguinte lógica:

```
1  let res = parimpar(4)
2
3  function parimpar(n) {
4    if (n % 2 == 0) {
5      return 'Par!'
6    } else {
7      return 'Ímpar!'
8    }
9  }
10
11 console.log(res)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

"C:\Program Files\nodejs\node.exe" ".\Módulo F\ex.012\funcao01.js"

Par!

Figura 78 - Explicação de Hoisting

11 Conclusão

Concluindo este documento introdutório sobre JavaScript, é importante ressaltar que as bases aqui apresentadas são fundamentais para uma compreensão sólida da linguagem. No entanto, para se tornar um desenvolvedor JavaScript mais completo e capaz de lidar com projetos mais complexos, é altamente recomendado aprofundar os estudos em diversas áreas-chave.

Primeiramente, o entendimento aprofundado de funções é essencial. Funções não apenas ajudam a organizar e reutilizar código, mas também são a base para entender conceitos avançados como escopo, closures e callbacks.

Além disso, compreender objetos em JavaScript é crucial. Os objetos são a espinha dorsal da linguagem e entender como trabalhar com eles, criar propriedades e métodos, e como manipular objetos é essencial para a programação eficaz em JavaScript.

A modularização do código é outra habilidade importante. A capacidade de dividir seu código em módulos reutilizáveis não só torna seu código mais organizado e fácil de manter, mas também facilita a colaboração em equipes de desenvolvimento.

O conhecimento de Expressões Regulares (RegEx) é valioso para lidar com manipulação de texto e validação de entrada do usuário de forma eficiente.

Entender o formato JSON (JavaScript Object Notation) e como interagir com ele é crucial, especialmente considerando que muitas APIs retornam dados nesse formato.

AJAX (Asynchronous JavaScript and XML) é uma tecnologia fundamental para criar aplicações web interativas e dinâmicas. Compreender como fazer requisições assíncronas para o servidor e lidar com as respostas é uma habilidade indispensável para desenvolvedores web modernos.

Por fim, explorar o ambiente de execução JavaScript fora do navegador é essencial. Node.js é uma plataforma que permite executar JavaScript no servidor, o que possibilita a criação de aplicações web escaláveis e eficientes.

Em resumo, o aprendizado contínuo e a exploração desses tópicos mencionados - funções, objetos, modularização, RegEx, JSON, AJAX e Node.js - são essenciais para se tornar um desenvolvedor JavaScript habilidoso e preparado para enfrentar desafios complexos no mundo do desenvolvimento web e de software.