

5 - JAVASCRIPT I //ASSUNTO DO MÓDULO

5.1 - COMPREENDER A HISTÓRIA DO JAVASCRIPT E COMPREENDER SUAS FUNÇÕES BÁSICAS //NOME ACIMA DOS ICONES DA AULA ABAIXO DO NÚMERO

5.1.1 - História e função do JavaScript//DOS TÍTULOS DO HYPERTEXTTO

5.1.1.1 - Origem da linguagem JavaScript

Em 1995, Brendan Eich (1961-) criou a linguagem de programação JavaScript. Originalmente, ela era denominada "Live Script" pela empresa Netscape. JS ou JavaEscript é o codenome para a implementação do ECMA Script. European Computer Manufacture's Association-ECMA; JavaScript não tem nada haver com Java. Tipagem fraca. Tipagem dinâmico traz problemas. 2012 foi lançado pela Microsoft o TipeScript, mais seguro.

O seu **objetivo inicial** era validar formulários HTML, porém, atualmente, JavaScript é bastante utilizada pelo **client-side**, ou seja, *uma linguagem executada no computador do usuário*.

5.1.1.2 - Função da Linguagem JavaScript

principal função da linguagem JavaScript é permitir ao programador *implementar itens mais complexos* em um site. Por exemplo:

Imagine que você acessa um site, insere seus dados e clica no botão "calcular". Então, ele calcula o seu Índice de Massa Corporal (IMC), retornando um valor no final. Essa tarefa foi uma função escrita em JavaScript.

5.1.1.3 - Características

A linguagem JavaScript independe de plataforma, pois seus comandos são interpretados pelo browser (navegador) do próprio usuário. Não fica atrelado ao sistema operacional. O seu navegador traduz para o seu sistema operacional. Com JavaScript, não é necessário fazer um código próprio para cada tipo de sistema. Um exemplo prático é o que acontece com aplicativos de celular que são programados duas vezes: uma para funcionar no sistema iOS (Apple) e outra no Android (Google).

5.1.1.4 - A trindade das linguagens Front-End

é composta pelas linguagens HTML, CSS e JavaScript. HTML é responsável pela estrutura da página, CSS, pela parte de edição da página e JavaScript, por toda a lógica envolvida na página web. Comparando com o corpo humano, **HTML é o esqueleto, CSS, a pele e JavaScript, o cérebro**.

5.1.2 - Identificar a finalidade da linguagem para o contexto web

javascript evoluiu tanto que funciona tanto no **Front-end** (lado cliente, executado na própria máquina do usuário) quando no lado back-end (o código é rodado no servidor)

JS é uma linguagem de servidor, um ambiente de programar no servidor é o Node.js, pode executar programas fora de ordem ajudando na escalonagem da aplicação.

2021 JS foi o mais utilizado, devido a linguagem padronizada e atualizada por meio da comunidade ativa de programadores, que criam bibliotecas. **JS não se limita a WEB**, Electron.js, possibilita criação de aplicação desktop. Código aberto.

5.1.2.1 - A importância do JavaScript em navegadores

Os navegadores são softwares que servem para acessar páginas da internet escritas em HTML.

Antigamente, os sites não interagiam muito com o usuário. Eles eram estáticos e, quando tinham alguma interação, demoravam para enviar uma resposta. Já imaginou? Em Old Web Today você pode ver como os sites eram em diferentes navegadores.

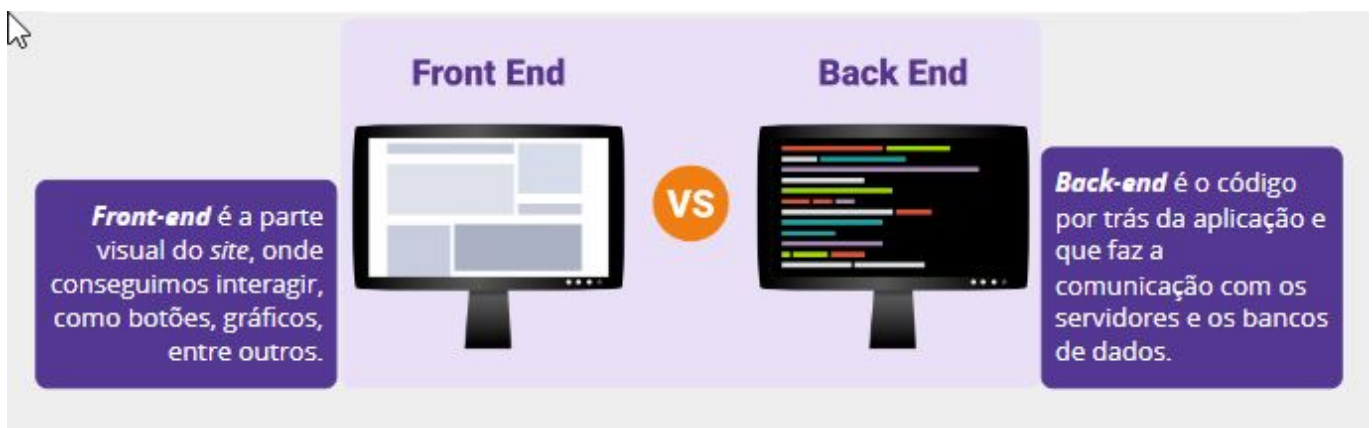
A linguagem JavaScript foi desenvolvida para facilitar a comunicação e a interação, deixando a navegação mais rápida e fácil.

Uma das **características** da linguagem JavaScript é **que ela é leve e interpretada**. Ou seja, o código é executado e o valor é retornado imediatamente.

Dessa forma, ela tem uma etapa a menos quando comparada às linguagens compiladas, nas quais o código é agrupado e, depois, executado pelo computador.

Navegador executa o próprio script sem necessidade de compilação. Os navegadores mais modernos conseguem utilizar uma técnica de compilação chamada just-in-time. Ela melhora o desempenho, fazendo com que o código seja executado mais rapidamente, pois ele só é executado quando necessário, ou seja, apenas quando o usuário faz alguma ação no site.

A linguagem JavaScript foi criada para as demandas de front-end, mas, devido ao avanço da tecnologia e às novas necessidades, surgiu a ideia de utilizá-la para a comunicação com o servidor, utilizando o Node.js para a programação back-end.



Front-end é a parte visual do site, onde conseguimos interagir, como botões, gráficos, entre outros.

Back-end é o código por trás da aplicação e que faz a comunicação com os servidores e os bancos de dados.

HTML funciona como o **esqueleto do site**, **CSS** como uma **roupa** (insere cores, fontes etc.) e a linguagem **JavaScript** serve **para provocar interações ou respostas**.

A JS é uma linguagem interpretada, pois não precisa do passo de compilação.

Uma barra de navegação interativa é um exemplo do uso do JavaScript, quando o usuário passa o mouse por cima de uma categoria com subcategorias (ex. Cursos) *Uma nova barra de navegação flutuante* é exibida logo embaixo da sua respectiva categoria. Este navegação flutuante, por sua vez, pode exibir uma segunda barra de navegação flutuante, posicionada ao lado da opção que "gerou ela", quando o mouse é passado por cima de um opção com outras "sub-opções" (ex. Cursos de aprimoramento).

Quando o mouse muda de posição para outra opção, a primeira barra de navegação flutuante some da página antes de uma nova navegação flutuante ser exibida, evitando sempre a sobreposição de barras de navegação. Além disso, se o mouse "sair de cima" de qualquer navegação flutuante, ou qualquer categoria que possua subcategorias, toda a "cadeia de barras de navegação" some também.

Embora seja possível mudar a opacidade de um elemento ou inclusive remover ele da página, de acordo com a interação do mouse, usando apenas CSS, esta é uma "reação" que aplicamos a um único elemento. Gerar mudanças no site, como por exemplo manipular elementos distintos daquele com o qual o mouse (ou o usuário) está interagindo não é um comportamento que podemos implementar usando apenas HTML e CSS.

5.1.3 - Diferenciar os ambientes existentes para desenvolvimento em JavaScript

5.1.3.1 - IDE's

Integral Development Environment-IDE, é um ambiente de desenvolvimento **com várias ferramentas que auxiliam** o programador na criação de seus códigos. Elas **são vinculadas a uma linguagem**, gerando acesso a várias funções específicas.

5.1.3.2 - Editores de código

Diferente das IDEs, os editores de código-fonte são ambientes de desenvolvimento que **NÃO** estão focados em uma linguagem específica.

Podem usar mais de uma linguagem, e /ou serem modificadas. É um editor de texto com superfunções.

5.1.3.3 - Como montar o seu ambiente?

A depender das suas necessidades(qual linguagem /qual complexidade).

[10 melhores editores online para JavaScript]<https://educationecosystem.com/blog/10-melhores-editores-online-para-javascript/>

[Lista de Plugins e Configurar JavaScript VSCode]<https://www.rtancman.com.br/javascript/configurar-vscode-para-javascript.html>

[10 ótimas extensões VSCode]<https://desenvolvimentoparaweb.com/javascript/visual-studio-code-javascript-extensoes-plugins/>

[Escreva limpo usando Prettier e ESLint]<https://medium.com/@pgivens/write-cleaner-code-using-prettier-and-eslint-in-vscode-d04f63805dcd>

CSS Peak, lista os estilos aplicados em cada classe ou id sem ter que ir a folha de estilos procurar. Só segurar Controle e passar o mouse por cima.

5.1.4 - VARIÁVEIS EM JS

5.1.4.1 - Tipos de variáveis

Aceita todos os anteriores. Deve-se informar o tipo, ao contrario do Python que já analisa o tipo sozinho. Pode-se usar array também

5.1.4.2 - Declaração de variáveis

Para declarar uma variável em JS **precisamos de 4 elementos**:

- A palavra reservada para o tipo da variável (var, let ou const)
- O nome da variável (ex. "num")
- O operador de atribuição ('=')
- O valor a ser atribuído (ex. 5)

```
var num = 5;
```

Obs. No JS é uma boa prática colocar um ponto e vírgula ';' no final de cada linha de código, mais ou menos como um ponto final no fim de cada frase em português.

5.1.4.3 - Var, Let e Const

Quais são as diferenças **entre os três tipos de variáveis no JS**, e como sabemos quando usar cada um deles? De forma breve, **'var'** foi a primeira palavra reservada para declarar variáveis no JS, porém, *ela caiu em desuso*.

Qual é então a diferença entre **'let'** e **'const'**? O termo const vem de "constante", e *usamos ele para guardar valores que não poderão ser alterados após sua declaração*. No exemplo a seguir, tentamos atribuir um novo valor a uma constante, mas o sistema mostrará uma mensagem de erro.

```
const num = 5;  
  
num = 10;
```

// Será disparada uma mensagem de erro, avisando que

// não podemos atribuir um novo valor a uma constante

Em contrapartida, valores salvos em variáveis do tipo **let** *podem ser alterados após sua declaração*:

```
let num = 5;
```

```
num = 10;

console.log(num);

// Imprimirá o valor 10 no terminal
```

Atenção! A função **console.log()** funciona da mesma forma que a **função print() em Python**. Entraremos em mais detalhe sobre o uso desse e outras funções nos próximos conteúdos. Atenção!! Atribuir valor é diferente de declarar a variável novamente. Para declarar usamos **var**, **let**, **const**.

[Diferença const var e let]<https://youtu.be/ZOx7iTnBqFQ>

var e let, tem uma diferença de escopo, var é escopo global, independente de onde é declarada ele funciona em toda a estrutura, **let só funciona dentro do bloco onde foi declarada**.

O escopo do const é igual ao do let, porém com a peculiaridade de que constantes não tem seu valor alterado.

[var, let e const – Qual é a diferença?]<https://www.freecodecamp.org/portuguese/news/var-let-e-const-qual-e-a-diferenca/>

Conteúdo abaixo tirado da página acima.

VAR, escopo é global quando uma variável var é declarada fora de uma função. Isso significa que qualquer variável que seja declarada com var fora de um bloco de função pode ser utilizada na janela inteira.

Variáveis de var podem ser declaradas de novo e atualizadas Isso significa que é possível fazer o seguinte dentro do mesmo escopo e não gerar um erro:

```
var greeter = "hey hi";
var greeter = "say Hello instead";
```

Um dos recursos que surgiu com o ES2015-ES6 foi a adoção de **let** e **const**. Antes era só **var**

Declarações com var tem escopo global ou escopo de função/local.

escopo é global quando uma variável var é declarada fora de uma função. Isso significa que qualquer variável que seja declarada com var fora de um bloco de função pode ser utilizada na janela inteira. **var tem escopo de função** quando é declarado dentro de uma função

Variáveis de var podem ser declaradas de novo e atualizadas.

```
var greeter = "hey hi";
var greeter = "say Hello instead";
```

#ou

```
var greeter = "hey hi";  
greeter = "say Hello instead";
```

Hoisting de var Hoisting é um mecanismo do JavaScript que faz com que as *declarações* de variáveis e de funções sejam *movidas para o topo* de seu escopo antes da execução do código.

```
var greeter = "hey hi";  
var times = 4;  
  
if (times > 3) {  
    var greeter = "say Hello instead";  
}  
console.log(greeter) // o resultado será "say Hello instead"
```

Assim, já que **times > 3** retorna true, greeter é redefinido como "say Hello instead". Embora isso não seja um problema se você quer, conscientemente, que greeter possa ser redefinido, passará a ser um problema quando você não perceber que uma variável greeter já havia sido definida antes.

Se você já usou greeter em outras partes do seu código, pode se surpreender com o resultado que vai obter. Isso provavelmente causará vários bugs no seu código. É por isso que let e const são necessários.

LET é, agora, a forma preferida de declaração de variáveis. Não é uma surpresa, já que ele é uma melhoria às declarações com var. Ele também resolve o problema de var do qual acabamos de tratar. Vamos ver a razão disso.

let tem escopo de bloco, Um bloco é uma porção de código cercado por {}. Um bloco vive dentro dessas chaves. Tudo o que estiver cercado por chaves é um bloco.

Assim, uma variável declarada com let em um bloco estará disponível apenas dentro daquele bloco. Vamos explicar isso com um exemplo:

```
let greeting = "say Hi";  
let times = 4;  
  
if (times > 3) {  
    let hello = "say Hello instead";  
    console.log(hello); // dirá "say Hello instead"  
}  
console.log(hello) // hello não está definido
```

Vemos que o uso de `hello` fora de seu bloco (as chaves dentro das quais a variável foi definida) retorna um erro. Isso ocorre porque as variáveis de `let` têm escopo de bloco.

let pode ser atualizado, mas não declarado novamente, Assim como `var`, uma variável declarada com `let` pode ser atualizada dentro de seu escopo. *Diferente de `var`, no entanto, uma variável `let` não pode ser declarada novamente dentro de seu escopo.* O código a seguir funciona:

```
let greeting = "say Hi";
greeting = "say Hello instead";

//Este outro código, no entanto, retornará um erro:
let greeting = "say Hi";
let greeting = "say Hello instead"; // erro: identificador 'greeting' já foi
declarado

//Porém, se a mesma variável for definida em escopos diferentes, não haverá erro:
let greeting = "say Hi";
if (true) {
  let greeting = "say Hello instead";
  console.log(greeting); // retornará "say Hello instead"
}
console.log(greeting); // retornará "say Hi"
```

Por que isso não retorna um erro? Porque as duas instâncias **são tratadas como variáveis diferentes, já que são de escopos diferentes**.

Este fato torna o `let` uma escolha melhor do que `var`. Ao usar `let`, você não precisa se preocupar se usou o nome para uma variável antes, já que a variável existe somente dentro daquele escopo.

Além disso, como uma variável não pode ser declarada mais de uma vez dentro de um escopo, o problema que ocorre com `var` que discutimos antes não acontece.

Hoisting de let Assim como as declarações com `var`, as feitas com `let` também sofrem o hoisting para o topo.

Diferentemente de `var`, porém, que é inicializado como `undefined`, a palavra-chave `let` não é inicializada. Assim, se você tentar usar uma variável `let` antes de sua declaração, terá um `Reference Error`.

Const Variáveis declaradas com `const` mantêm valores constantes. Declarações com `const` compartilham algumas semelhanças com as declarações com `let`.

Declarações com const têm escopo de bloco Assim como as declarações de `let`, as declarações de `const` somente podem ser acessadas dentro do bloco onde foram declaradas.

const não pode ser atualizado nem declarado novamente Isso significa que o valor de uma variável declarada com `const` se mantém o mesmo dentro do escopo. Ela não pode ser atualizada nem declarada

novamente. Desse modo, se declararmos uma variável com `const`, isso não será possível:

```
const greeting = "say Hi";
greeting = "say Hello instead";// erro: atribuição a uma variável constante.
//O código abaixo também não:

const greeting = "say Hi";
const greeting = "say Hello instead";// erro: identificador 'greeting' já foi
declarado
```

Cada declaração com `const`, portanto, deve ser inicializada no momento da declaração.

Esse comportamento **é bastante diferente quando falamos de objetos declarados com `const`**. Embora um objeto declarado com `const` não possa ser atualizado, **é possível atualizar as propriedades desse objeto**. Assim, podemos declarar um objeto com `const` dessa forma:

```
const greeting = {
  message: "say Hi",
  times: 4
}
//Não será possível fazer isso:

greeting = {
  words: "Hello",
  number: "five"
} // erro: atribuição a uma variável constante.
//Será possível, contudo, fazer isso:

greeting.message = "say Hello instead";
//Isso atualizará o valor de greeting.message sem retornar erros.
```

Hoisting de `const`, assim como as declarações de `let`, as de `const` **sofrem o hoisting para o topo** do escopo, **mas não são inicializadas**.

Bem, caso você não tenha visto as diferenças, aqui estão:

- As declarações de `var` tem escopo global ou de função, enquanto as declarações de `let` e de `const` têm escopo de bloco.
- variáveis de `var` podem ser atualizadas e declaradas novamente dentro de seu escopo. As variáveis de `let` podem ser atualizadas, mas não podem ser declaradas novamente. As variáveis de `const` não podem ser atualizadas nem declaradas novamente.
- Todas elas passam por hoisting para o topo de seu escopo. Porém, enquanto variáveis com `var` são inicializadas com `undefined`, as variáveis com `let` e `const` não são inicializadas.
- Enquanto `var` e `let` podem ser declaradas sem ser inicializadas, `const` precisa da inicialização durante a declaração.

5.2 - OPERADORES EM JS

5.2.1 - Operadores em JS

Os **operadores aritméticos** (soma, subtração, multiplicação, divisão e módulo) e **relacionais** (maior, menor, maior ou igual, menor ou igual) **são exatamente iguais em JS e Python**.

5.2.1.1 - Operadores aritméticos

Operação	Operador
Soma	+
Subtração	-
Multiplicação	*
Divisão	/
Módulo	%
Maior que	>
Menor que	<
Maior ou igual que	>=
Menor ou igual que	<=

5.2.1.1 - Operadores lógicos

Operação	Operador (Python)	Operador (JS)
Conjunção	and	&&
Disjunção	or	
Negação	not ou !	!

```
(10 > 5) && (2 >= 7)

// O resultado é 'false'


(10 > 5) || (2 >= 7)

// O resultado é 'true'


!(10 > 5)
```

```
// O resultado é 'false'
```

[REIS, Ricardo. Operadores Lógicos (Logical Operators) JavaScripts]<https://ricardo-reis.medium.com/operadores-lógicos-logical-operators-b0687819d1a5>

[Dev Aprender. Javascript Tutorial 14 - Operadores Lógicos (Operadores)]<https://www.youtube.com/watch?v=Vhw8AaiSUjU>

5.2.2 - Estruturas Condicionais em JS

5.2.2.1 - Estruturas Condicionais em JS

Para escrever estruturas condicionais em JS precisamos de:

- Pelo menos uma palavra reservada (**if**, **else**)
- Uma condição a ser testada entre parênteses (ex. (3 > 1) 😊)
- Um bloco de código a ser executado entre chaves '{ }'

```
let nota = 9;

if(nota >= 8){
    console.log("Ótimo trabalho!");
}
```

Observação. Já que as chaves determinam o começo e fim do bloco de código a ser executado, a indentação do bloco não é obrigatória mas ainda é uma boa prática, que visa facilitar a leitura do código.

Para definirmos blocos de código a serem executados caso a condição não for verdadeira, usamos a palavra reservada **else** e as chaves, porém, não é mais necessário o uso de parênteses:

```
let nota = 7;

if(nota >= 8){
    console.log("Ótimo trabalho!");
}
```

```
} else {  
  
    console.log("Precisa melhorar");  
  
}
```

definir **duas ou mais condições** a serem avaliadas, usamos a junção de ambas as palavras definidas (else if), um par de parênteses com a nova condição, e o bloco de código a ser executado entre chaves. Vale lembrar que **nosso código será lido de cima para baixo**, portanto, *a ordem em que as condições são escritas faz diferença*, e o bloco de código **else precisa sempre ser o último**:

```
let nota = 6;  
  
if(nota >= 8){  
    console.log("Ótimo trabalho!");  
} else if(nota >= 6){  
    console.log("Bom trabalho");  
} else {  
    console.log("Precisa melhorar");  
}
```

5.2.2.2 - Condicionais e booleanos em JS

As condições das nossas estruturas condicionais podem incluir números (como nos exemplos anteriores), incluir strings:

```
let meuInstrumento = "violão";  
  
if (meuInstrumento == "piano"){  
    console.log("Você é um pianista!");  
} else {
```

```
    console.log("Você não é um pianista");

}

// Imprimirá 'false' pois meuInstrumento não é 'piano'

let finDeSemana = false;

if(finDeSemana == true){

    console.log("Vai descansar!");

} else {

    console.log("Temos que trabalhar");

}

// Imprimirá 'Temos que trabalhar'
```

JavaScript nos permite escrever a condição entre parênteses (`finDeSemana == true`) de uma **forma mais simples**: apenas colocando o nome da variável.

```
let finDeSemana = false;

if(finDeSemana){

    console.log("Vai descansar!");

} else {

    console.log("Temos que trabalhar");

}
```

```
// Imprimirá 'Temos que trabalhar'
```

Sempre que trabalhamos com variáveis que guardam valores booleanos em JS podemos verificar sua “veracidade” apenas chamando o nome da variável. Já se quisermos verificar se o valor do booleano é falso, podemos usar o operador de negação (!) antes do nome da variável:

```
let finDeSemana = false;

if(!finDeSemana){

    console.log("Temos que trabalhar.");

} else {

    console.log("Vai descansar!");

}

// Imprimirá 'Temos que trabalhar'
```

Perceba que como agora o primeiro bloco condicional está verificando se a variável `finDeSemana` é falsa, a mensagem impressa é o que fazer num dia de semana. Já como o segundo bloco de código (no `else`) que é executado caso `finDeSemana` for verdadeiro, imprime agora a mensagem relativa ao final de semana. Para efeitos práticos, este código funciona exatamente igual aos dois códigos anteriores. Este é um exemplo de como podemos chegar no mesmo resultado escrevendo algoritmos e códigos distintos

[MDN Web Docs. Tomando decisões no seu código]https://developer.mozilla.org/pt-BR/docs/Learn/JavaScript/Building_blocks/conditionals

[DevMedia. JavaScript if/else: criando scripts com estruturas condicionais]<https://www.youtube.com/watch?v=8UXQ6S0KURk>

5.2.3 - Loops em JS

Há várias, porém o **for** será estudado. Ele precisa de três parâmetros. Permite usar variável dinâmica dentro do bloco.

A estrutura inicial para escrever um for loop em JS consiste da palavra reservada **for**, os parâmetros do loop entre parênteses, e o código a ser executado entre chaves.

```
for(/*Parâmetros do loop*/ ){  
    // Código a ser executado "em loop"  
}
```

5.3.1.1 - Sintaxe do "for loop"

Dentro dos parênteses escreveremos os três parâmetros necessários, separados por ponto e vírgula.

- **Variável contadora**

Declaramos uma variável e um valor inicial para ela

É comum definir o nome da variável como 'i' de 'índice'

Exemplo: let i = 0;

- **Condição de parada**

Expressão avaliada antes do início de cada repetição

Quando a expressão for avaliada como falsa a estrutura de repetição chega ao fim

Exemplo: i < 10;

- **Incremento ou decremento**

Determina o aumento ou decremento a ser aplicado à variável contadora no fim de cada repetição

JavaScript nos fornece a seguinte sintaxe simplificada para adicionar 1 à variável: i++

Esta sintaxe simplificada é equivalente a: (i = i + 1). Isto é, atribuir à variável 'i' o "valor atual dela mais um"

Aplicando os três parâmetros à nossa estrutura inicial, chegamos no seguinte código:

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

No exemplo anterior, nosso código imprimirá os números de 0 a 9, pois a condição de parada é a variável contadora ser menor que o número 10.

5.3.1.2 - Percorrer um array

Podemos usar os for loop para percorrer arrays da mesma forma que o fazemos em Python. A principal diferença é o uso da propriedade length do array. Length (do inglês “cumprimento”) é uma propriedade que nos retorna a quantidade de elementos num array. Para usar ele **basta chamar a variável que guarda o array**, seguida de um ponto '.', e a palavra reservada **length**.

```
let letras = ['a', 'b', 'c'];

console.log(letras.length);

// Imprimirá '3' pois o array 'letras' tem três elementos
```

Podemos usar a propriedade length **para percorrer arrays** em JS sem nos preocupar pela quantidade de elementos. Para fazer isso, devemos inclui-lo como condição de parada do nosso for loop da seguinte forma:

```
let letras = ['a', 'b', 'c'];

for(let i = 0; i < letras.length; i++){

    console.log(letras[i]);

}

// Imprimirá 'a', 'b', e 'c'
```

estamos usando a variável dinâmica 'i' para acessar cada elemento do array, chamando o array (neste caso, letras) seguido da variável contadora entre colchetes. Isto é equivalente a imprimir “letras[0]” na primeira volta, “letras[1]” na segunda volta, e “letras[2]” na terceira e última volta.

[Estrutura de repetição - Percorrer um Array]<https://youtu.be/wiUUASSieOE>

Percorrendo um Array usando como parâmetro o tamanho do array.

```
var arr = [1,2,3,4];  
  
for(var j = 0; j < arr.length; j++) {  
    console.log(arr[j]);  
}
```

[MDN Web Docs]<https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Statements/for>

[Curso JavaScript #21 - for (estrutura de repetição - loop)]<https://www.youtube.com/watch?v=wiUUASSieOE>

[Exercício-CodePark03]<https://onecompiler.com/javascript/3znt73w2g>

Comentários ao exercício

Em primeiro lugar, definimos a estrutura do 'for loop' para acessar cada um dos elementos do array, usando o cumprimento de `numerosDaSorte` como condição de parada. Assim, dentro do bloco de código que será executado entre as chaves podemos acessar de forma dinâmica cada um dos elementos do array usando a variável contadora como índice (ex. `numerosDaSorte[i]`).

Em segundo lugar, usamos uma estrutura condicional com as palavras reservadas 'if', 'else if', e 'else'. Cada uma com seu respectivo bloco de código a ser executado, imprimindo uma das três possíveis frases para cada número. Na condição do 'if' verificamos se o número é par, usando o operador módulo (ex. `num % 2 == 0`), e usamos o operador de conjunção '&&' para verificar se o número é também menor do que 50. Já no bloco 'else if' verificamos apenas se o número é menor do que 50.

Finalmente, como sabemos que todos os números que não passarem nos dois testes lógicos são necessariamente menores do que 50 e, como neste caso não precisamos fazer nenhuma outra avaliação para os números menores do que 50, usamos o bloco 'else' para imprimir a última frase.

5.3 - FUNÇÕES EM JS

5.3.1 - Funções em Java Script

podem ou não receber argumentos elas podem ter um return podem incluir todos os conceitos estudados previamente dentro de seus blocos de código (arrays, declaração de variáveis, estruturas condicionais, estruturas de repetição, e até outras funções!).

5.3.1.1 - Funções regulares

Declaração:

- A palavra reservada 'function'
- O nome da função
- Parênteses para receber parâmetros (mesmo se não for receber, é necessário escrevê-los)

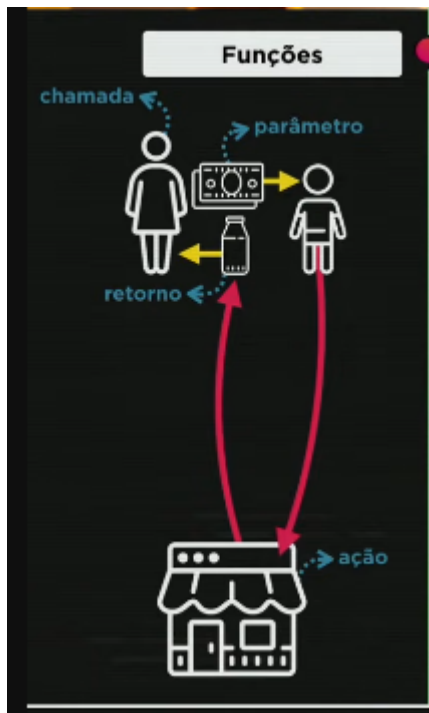
- Chaves com o bloco de código a ser executado quando a função for chamada

```
function cumprimentar(){  
    console.log("Boas-vindas!")  
}  
  
cumprimentar();  
  
// Imprimirá "Boas-vindas!"  
  
function multiplicar(num1, num2){  
    return (num1 * num2);  
}  
  
multiplicar(3, 7);  
  
// Imprimirá 21, que é o resultado de (3 * 7)
```

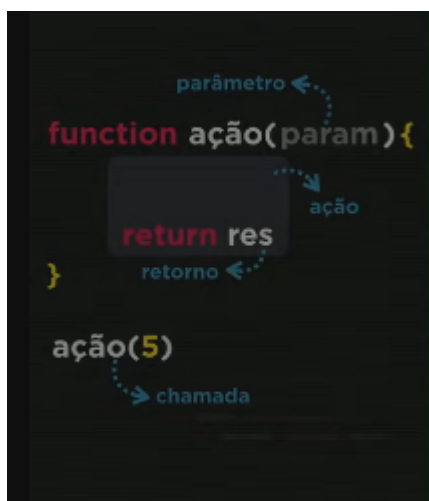
lembre-se Lembre que **para executar o bloco de código de qualquer função é necessário declarar ela**, e depois chamá-la, escrevendo apenas o nome da função seguida de parênteses (com argumentos, caso precisar deles).

[Funções JavaScript]<https://youtu.be/mc3TKp2Xzhl>

Comentários ao vídeo



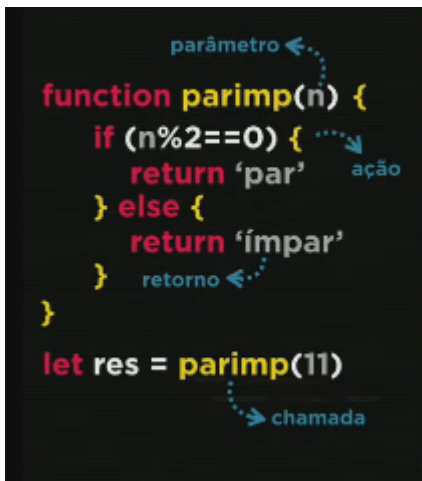
Funções são ações executadas assim que são chamadas, ou em decorrência de algum evento; Funções podem receber parâmetros e retornar resultado;



Pode haver mais de uma declaração de return dentro de uma `function` porém só pode retorna apenas uma.

Deve-se incluir a função dentro de uma variável para ser "chamada" e executada.

```
function parimp(n) {
  if (n%2==0) {
    return 'par'
  } else {
    return 'ímpar'
  }
}
let res = parimp(11)
```



```

function parimp(n) {
  if (n%2==0) {
    return 'par'
  } else {
    return 'ímpar'
  }
}

let res = parimp(11)

```

Diagram illustrating the execution of the `parimp` function:

- `parâmetro` points to the parameter `n`.
- `ação` points to the `return` statement inside the `if` block.
- `retorno` points to the `return` statement outside the `if` block.
- `chamada` points to the function call `parimp(11)`.

```

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

```

Atenção! Ficar esperto quando se informa que haverá por exemplo dois parâmetros e só é informado 1. O output será NaN (Not a Number) Para resolver isso, colocasse `=0` ao lado do parâmetro dizendo que se ele não for informado terá valor zero.

```

1 function soma(n1=0, n2=0) {
2   return n1 + n2
3 }
4
5 console.log(soma(7))

```

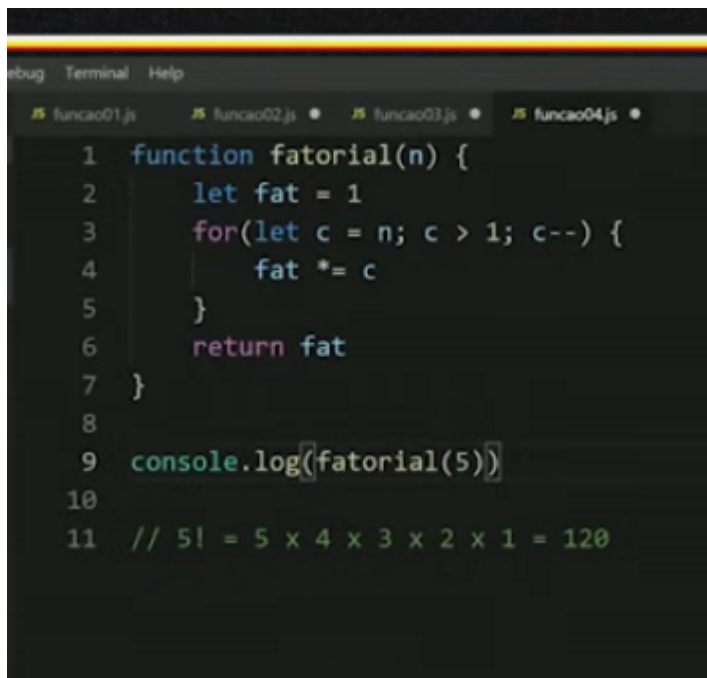
Pode-se incluir uma Função dentro de uma variável(FUNÇÕES ANONIMAS)

```

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

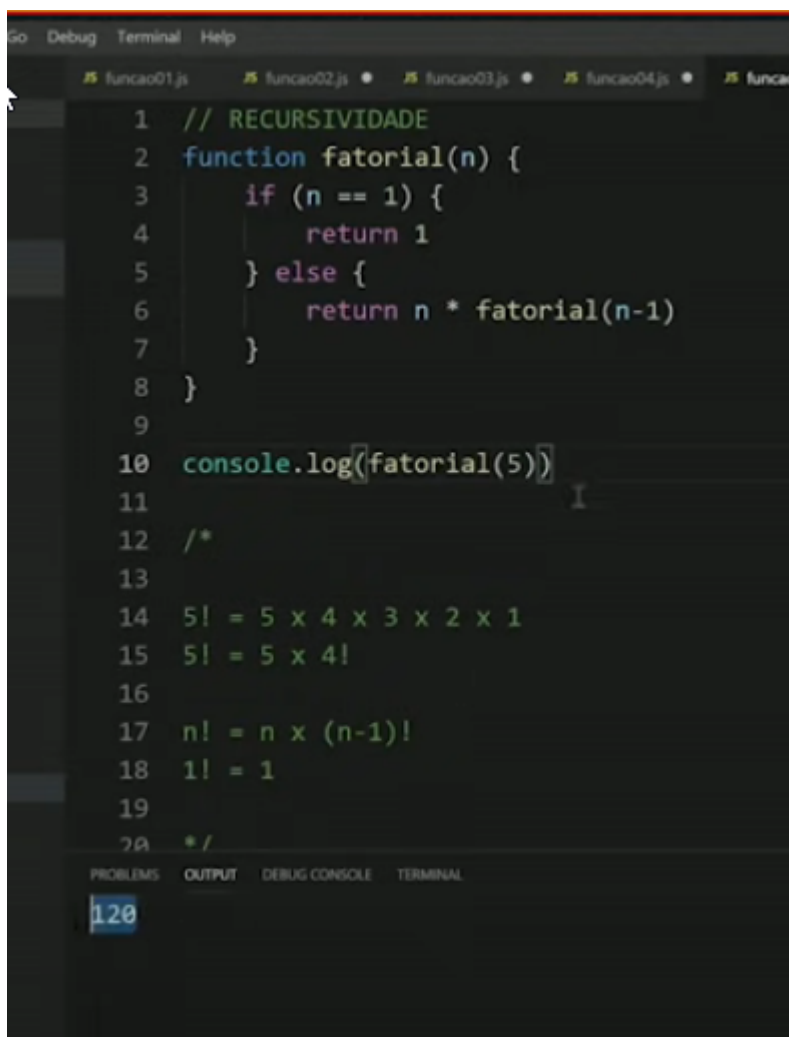
```

Calculando um fatorial



```
1 function fatorial(n) {  
2     let fat = 1  
3     for(let c = n; c > 1; c--) {  
4         fat *= c  
5     }  
6     return fat  
7 }  
8  
9 console.log(fatorial(5))  
10  
11 // 5! = 5 x 4 x 3 x 2 x 1 = 120
```

Pode-se fazer a mesma coisa usando RECURSÃO (Quando uma função chama outra função)



```
1 // RECURSIVIDADE  
2 function fatorial(n) {  
3     if (n == 1) {  
4         return 1  
5     } else {  
6         return n * fatorial(n-1)  
7     }  
8 }  
9  
10 console.log(fatorial(5))  
11  
12 /*  
13  
14 5! = 5 x 4 x 3 x 2 x 1  
15 5! = 5 x 4!  
16  
17 n! = n x (n-1)!  
18 1! = 1  
19  
20 */
```

120

o fatorial de 5 é $5 \times 4 \times 3 \times 2 \times 1$, assim como pode ser $5 \times$ fatorial de 4

a função criada está chamando ela mesma dentro dela.

5.3.1.2 - Funções anônimas

Estas funções **não possuem um nome quando declaradas**, e são geralmente atribuídas a uma variável que guarda a função como seu valor.

```
// Função regular
//preicisa da palavra reservada `function`. Parâmetros entre parênteses e
separados por vírgula. E precisa de `return`
function somar(a, b){

    return (a + b);

}

// Função anônima - Essas funções NÃO TEM um nome definido entre a palavra
reservada "function" e o par de parênteses. Mas podem ser atribuídas a uma
variável.

const adicionar = function(a, b){

    return (a + b);

}

const somar = function(a, b){

    return (a + b);

}

somar(5, 9);

// Imprimirá 14, que é o resultado de (5 + 9)
//Para chamar uma função anônima, basta chamar o nome da variável que a guarda,
seguida de um par de parênteses (com argumentos, caso precisar deles)
```

5.3.1.3 - Arrow functions (opcional)

JavaScript nos fornece mais um tipo, mais moderna das funções anônimas. Pensadas em simplificar. Escritas numa única linha de código.

```
// Função anônima declarada de forma tradicional

const seguinteNum = function(n){

    return (n + 1);

}
```

```

}

// Arrow function

const proximoNum = (n) => {

    return (n + 1)

}
//A **primeira** diferença que percebemos é que **não precisamos da palavra reservada **function****. A **segunda** diferença, é que *escrevemos* “uma seta gorda” com os símbolos ‘=’ e ‘>’ **entre os parâmetros e o bloco de código**

// Arrow function (MAIS SIMPLIFICADA)

const proximoNum = n => return (n + 1)
//percebemos mais duas mudanças. Em primeiro lugar, quando temos um único parâmetro, não precisamos dos parênteses na hora de declarar a arrow function. Em segundo lugar, quando escrevemos o bloco de código na mesma linha em que temos a seta da arrow function, não precisamos das chaves para encapsular o bloco de código a ser executado

// Arrow function (SIMPLIFICADO DO SIMPLIFICADO)

const proximoNum = n => (n + 1)

//Da mesma forma que com as chaves, quando temos o bloco de código na mesma linha de código que a seta não precisamos do return

const vezesCinco = num => num*5
//não precisamos da palavra reservada "function", por ser uma arrow. Por ter apenas um parâmetro não precisamos de parênteses. E podemos dispensar as chaves e o return por estar tudo escrito numa única linha.

```

RESUMO

CONDIÇÃO (para uso)	SIMPLIFICAÇÃO
nenhuma	Não usar a palavra reservada function
nenhuma	Incluir uma seta gorda => entre os parâmetros entre parênteses
Um único parâmetro	Não precisamos encapsular o parâmetro entre parênteses

CONDIÇÃO (para uso)

Bloco de código na mesma linha que os parâmetros e a "seta"

SIMPLIFICAÇÃO

Não precisamos das **chaves**, nem da palavra reservada **return**

VALE LEMBRAR

O uso de arrow functions é completamente opcional. É perfeitamente possível escrever as mesmas funções de forma anônima, ou de forma regular. Contudo, o uso das arrow function tem ficado muito popular entre a comunidade de desenvolvedores de JS, e por isso é importante sabermos pelo menos reconhece-las e interpretá-las.



<https://youtu.be/S5Mn0qQzJ-0>

Comentários ao vídeo Af, apartir do ES6, são SEMPRE funções ANÔNIMAS. Trata o **this** de uma forma totalmente diferente.

tudo após a fatArrow é assumido como **return**; quando tem mais de uma função tem que colocar as chaves;

ArrowFunctions redefinem o **this**, this no normal é definido onde é chamado. nos Arrows, é definido no local que é criado. //é isso que entendi.

Material Complementar

[Funções no JavaScript]<https://encontreseudocodigo.com.br/aprenda-javascript/aprenda-funcoes-no-javascript/>

[Curso em Vídeo. Funções - Curso JavaScript #16]<https://www.youtube.com/watch?v=mc3TKp2Xzhl>

[Arrow Functions vs. Functions em JavaScript]<https://www.youtube.com/watch?v=S5Mn0qQzJ-0>

5.3.2 - Conexão com HTML

Existem duas formas de inserir código JavaScript nas nossas páginas web:

1. usando uma **tag específica** para JS no arquivo HTML.
 1. `<script></script>`, tudo inscrito entre essas tags devem seguir as regras de sintaxe.
2. criando um **arquivo separado** com todos nossos códigos JS, e depois "conectar" esse arquivo com nosso arquivo HTML.
 1. Usamos o mesmo método do CSS(criar um arquivo separado e conectalo ao html via link).

1. Primeiro, temos as tags de abertura e fechamento `<script> </script>` sem conteúdo algum entre delas, pois não estaremos usando elas para escrever código JS, e sim para conectar nosso arquivo HTML com um arquivo de extensão .js
2. Depois, incluímos o atributo `src=""` que vem de "source" ("origem" em inglês), que recebe uma string com a rota para nosso arquivo JS. Neste caso, a string começa com um ponto e barra './' indicando que procuraremos pelo arquivo na mesma pasta onde está o arquivo HTML, e depois da barra incluímos o nome do arquivo, resultando em:
`src="./script.js"`
3. Finalmente, incluímos o atributo `defer`, que faz com que o navegador *execute o arquivo JS apenas depois que o arquivo HTML tenha sido baixado e analisado*.

A ordem em que os arquivos são executados **pode não parecer importante** neste momento, porém, nas próximas aulas entenderemos melhor a importância do atributo "defer" e de carregar nosso código JavaScript depois do conteúdo HTML.

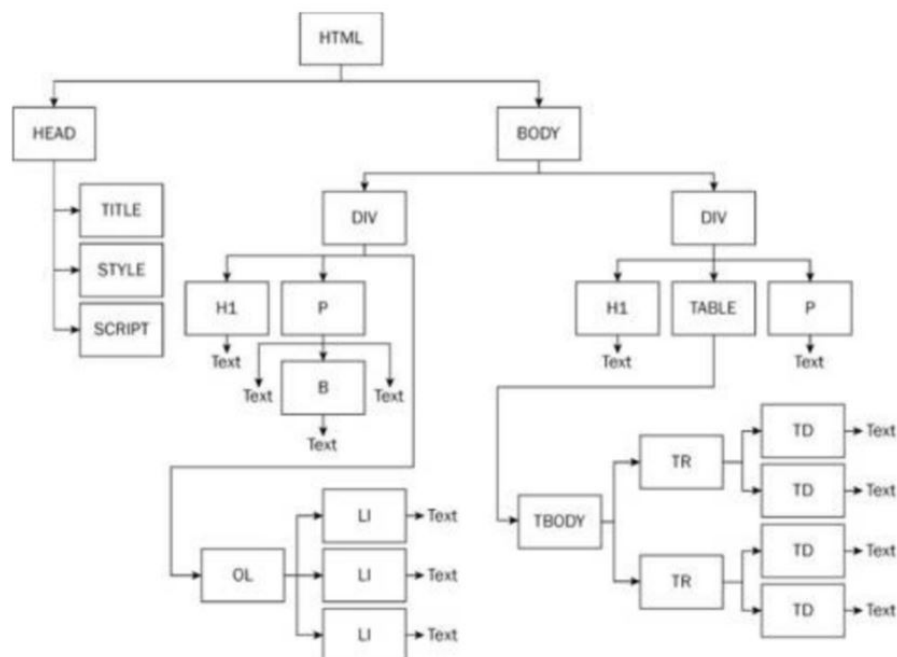
[TableLess. Inserindo JS]<https://tableless.github.io/iniciantes/manual/js/inserindo-js.html>

[W3Schools. HTML `<script>` defer Attribute]https://www.w3schools.com/tags/att_script_defer.asp.

5.4 - SELETORES DOM EM JS

5.4.1 - O que é DOM?

O DOM, das siglas em inglês "Document Object Model" (ou "Modelo de Objeto do Documento" em português), é um modelo que representa os elementos exibidos numa página web. **A ligação do DOM com nossos arquivos HTML é de dupla mão:** quando um elemento HTML é criado, uma representação dele no DOM é criado, e se alteramos alguma representação no DOM seu respectivo elemento HTML sofrerá as mesmas alterações no navegador. E quem nos ajudará a manipular o DOM? Você provavelmente já sabe a resposta: JavaScript!



Representação dos elementos HTML no DOM

5.4.2 - Acessando a DOM por ID e Classe

Para acessar os elementos do nosso arquivo HTML usaremos o objeto `document` criado pelo DOM. Este objeto **tem uma série de propriedades** que nos retornam informações sobre nossa página, como por exemplo a URL, ou os cookies.

5.4.2.1 Setup de arquivos

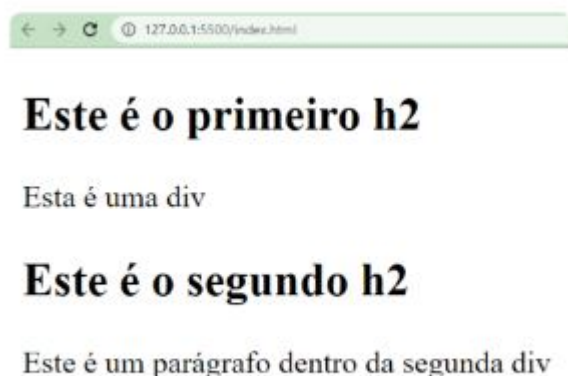
Para entender melhor como podemos acessar o DOM, comecemos criando uma nova pasta chamada **seletores-dom**, criemos o arquivo **index.html** com a estrutura base HTML, e adicionemos nele os seguintes elementos dentro da tag `body`:

```

<body>
  <h2 id="titulo">Este é o primeiro h2</h2>
  <div class="texto-simples">Esta é uma div</div>
</div>
  <h2>Este é o segundo h2</h2>
  <p class="texto-simples">Este é um parágrafo dentro da segunda div</p>
</div>
</body>
  
```

Como você pode ver temos **5 elementos HTML**. No "primeiro nível" temos um elemento h2, e duas divs. No "segundo nível" temos os "filhos" da segunda div: um segundo elemento h2 e um elemento de parágrafo. Observe que há um id com o valor "titulo" ao primeiro elemento h2, e a class "texto-simples" à primeira div e ao elemento de parágrafo.

Como não aplicamos nenhuma estilização à nossa página, se abrirmos nosso projeto no navegador com a ferramenta GoLive da extensão Live Server, ela deveria ter o seguinte aspecto:



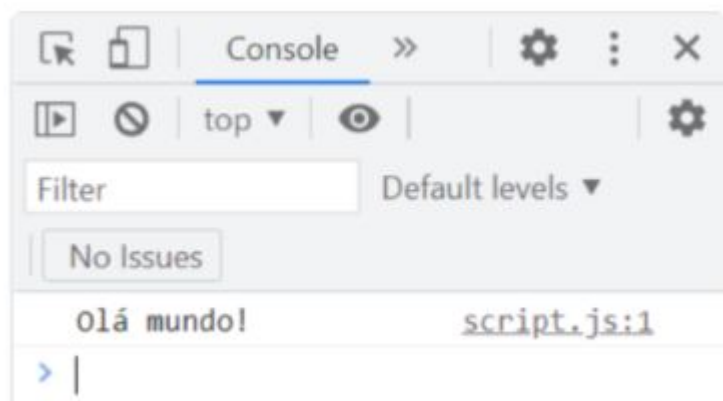
Criemos agora o arquivo script.js e conectemos ambos arquivos adicionando a seguinte tag script dentro da tag head do nosso arquivo index.html com o atributo defer:

```
<head>
  ...
  <script src="script.js" defer></script>
  <title>Seletores DOM</title>
</head>

<!-- Obs. Os três pontos representam as tags meta
-->
```

Para testar que a conexão foi feita corretamente, voltemos no arquivo script.js e imprimamos a frase "Olá mundo!" com a seguinte linha de código:

```
console.log("Olá mundo!");
```



Caso não consiga ver a mensagem no terminal, verifique que o nome do arquivo ("script.js") e o atributo src da tag script no arquivo index.html estão escritos da mesma forma. Outro possível motivo para não exibir a mensagem e não ter salvo as alterações em ambos arquivos

5.4.3 - Métodos de acesso dos elementos da DOM

Usaremos **quatro métodos** (funções guardadas em um objeto) para acessar os elementos da DOM. Os dois primeiros são:

Propriedade/Método	Descrição
getElementById()	Retorna o elemento que tem o ID com o valor específico
getElementsByClassName()	Retorna um HTMLCollection com todos os elementos que contem a class especificada.

No arquivo .js incluir o código `const titulo = document`. Delcaremos a variável que guardará o retorno do primeiro método, e atribuímos a ele o objeto document

Vale lembrar que dois elementos não devem ter o mesmo valor de ID numa mesma página, portanto, o método `.getElementById()` procura apenas um elemento. Para acessar vários elementos de uma só vez? Nossa primeira opção é usar o método `.getElementsByClassName()`, passar como argumento a string "texto-simples", e guardar o retorno numa variável. Usemos o `console.log()` para imprimir essa variável:

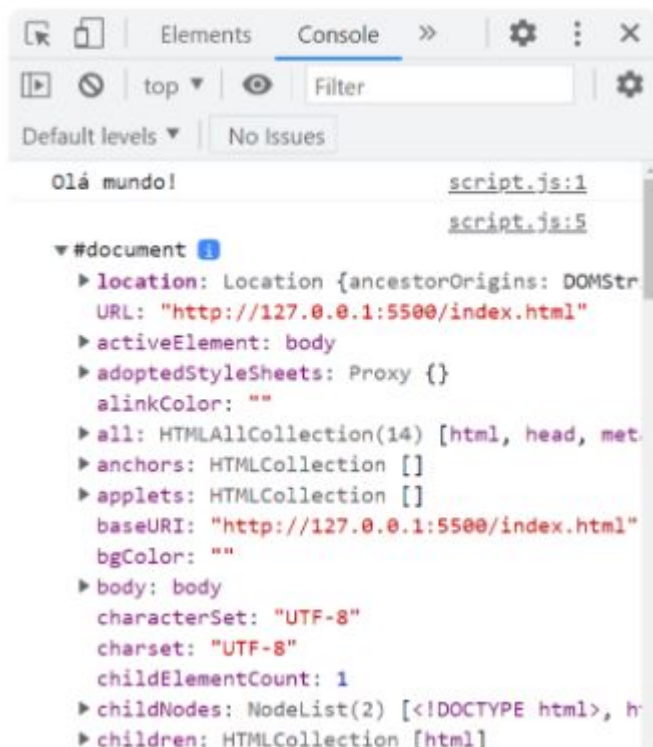
```
const textos = document.getElementsByClassName("texto-simples");  
  
console.log(textos);
```

Dessa vez, se conferirmos o terminal, teremos um resultado um pouco diferente: um HTMLCollection. As **HTMLCollection são semelhantes (porém, não iguais) a arrays**. Ao lado do termo HTMLCollection podemos ver um número 2 entre parênteses, indicando que é uma lista com dois elementos. Podemos ver também ambos os elementos entre chaves e separados por uma vírgula (seguindo a sintaxe de um array): uma div e um parágrafo com a classe "texto-simples".

Podemos acessar um por um os elementos da HTMLCollection da mesma forma como acessaríamos elementos de um array: **escrevendo o nome da variável que guarda a lista, e passando o índice de cada elemento entre chaves**. Se quisermos acessar, por exemplo, o primeiro elemento da lista, usáramos a seguinte linha de código:

```
console.log(textos[0]);
```

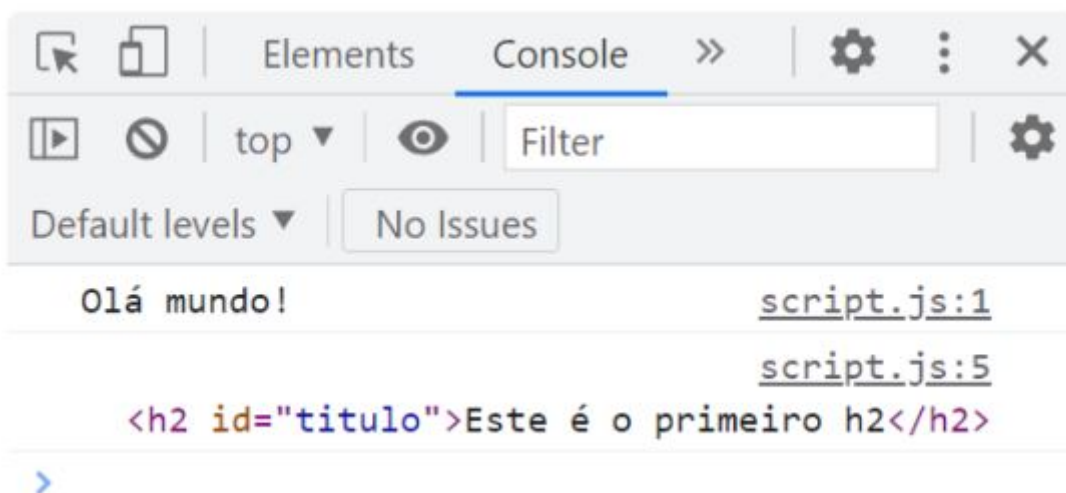
Se imprimirmos a variável `titulo` neste ponto, **podemos ver todas as informações que guarda o objeto document**.



5.4.3.1 - Acessando um elemento da DOM

Contudo, a gente não quer salvar todas essas informações na nossa **variável** `titulo`, senão **apenas o elemento h2 com id "titulo"**. Para isso, usaremos o método `.getElementById()` do objeto `document` e passaremos como argumento a string `"titulo"`.

```
const titulo = document.getElementById("titulo");
console.log(titulo);
```



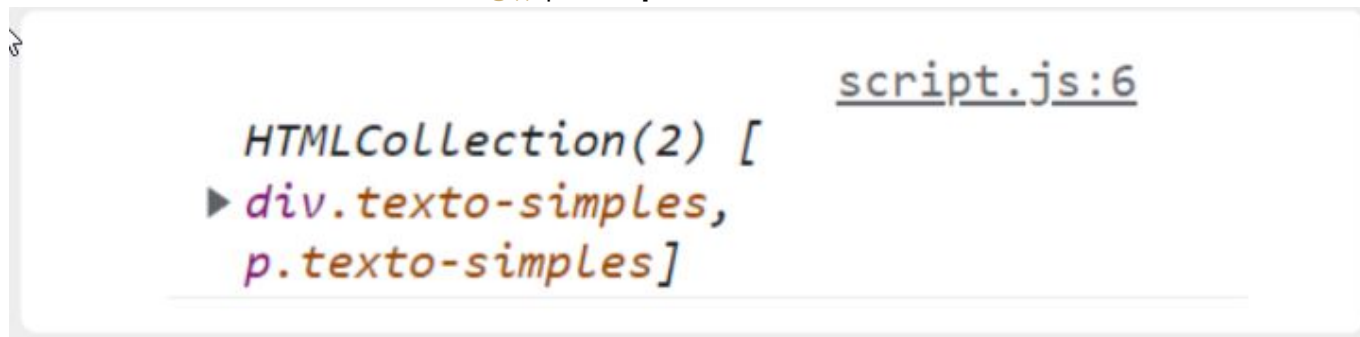
LEMBRE-SE

Vale lembrar que dois elementos não devem ter o mesmo valor de ID numa mesma página, portanto, o

método `.getElementById()` procura apenas um elemento.

5.4.3.2 - Acessando MAIS DE UM elemento da DOM

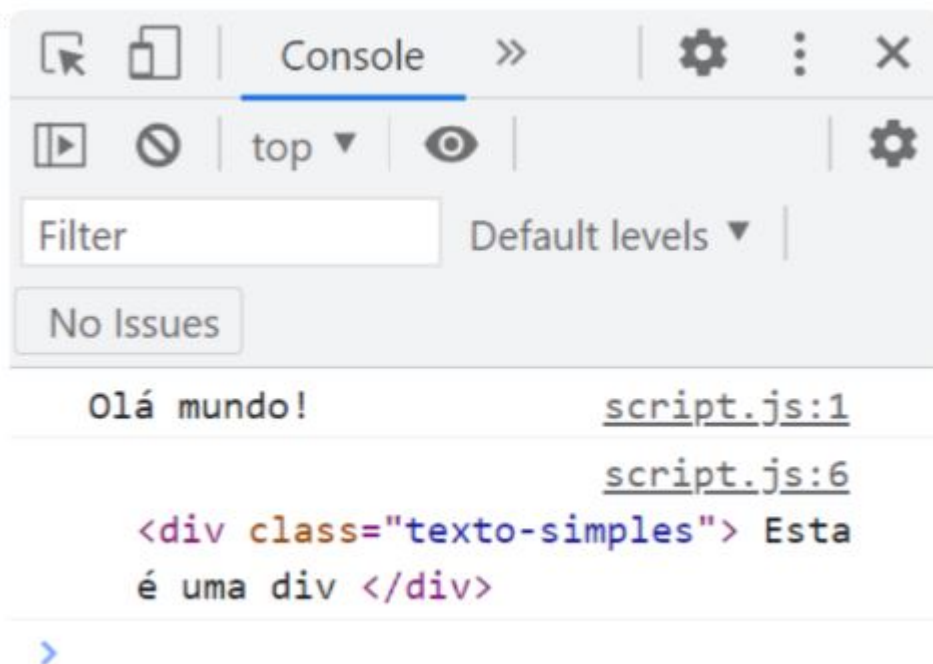
E se quisermos acessar **vários elementos de uma só vez**? Nossa primeira opção é usar o método `.getElementsByClassName()`, passar como argumento a string `"texto-simples"`, e **guardar o retorno numa variável**. Usemos o `console.log()` para **imprimir essa variável**:



Ao conferirmos o terminal, teremos um resultado um pouco diferente: um `HTMLCollection`. As `HTMLCollection` são semelhantes (porém, não iguais) a arrays. Ao lado do termo `HTMLCollection` podemos ver um número 2 entre parênteses, indicando que é uma lista com dois elementos. Podemos ver também ambos os elementos **entre chaves** e separados por uma vírgula (seguindo a sintaxe de um array): uma `div` e um parágrafo com a classe `"texto-simples"`.

Podemos **acessar um por um os elementos** da `HTMLCollection` da mesma forma como acessaríamos elementos de um array: escrevendo o nome da variável que guarda a lista, e passando o índice de cada elemento entre chaves. Se quisermos acessar, por exemplo, o primeiro elemento da lista, usaríamos a seguinte linha de código:

```
console.log(textos[0]);
```



5.4.3 - Acessando a DOM com seletores CSS

Os métodos anteriores são úteis se quisermos acessar elementos com id, ou com a mesma classe, porém, às vezes precisamos fazer seleções mais específicas. Uma solução é atribuir IDs e classes a todos os elementos que quisermos acessar, mas muitos desenvolvedores preferem usar os próprios seletores CSS para não poluir o arquivo HTML com um monte de atributos desnecessários ou redundantes.

Propriedade/Método	Descrição
querySelector()	Retorna o primeiro elemento no documento. Pode-se utilizar seletores CSS, "." para Classe e "#" para ID.
querySelectorAll()	Retorna uma NodeList com todos os elementos no documento que seguem a especificação de um seletor CSS

Dessa vez, queremos acessar o segundo elemento h2 da nossa página. Se você conferir o arquivo index.html perceberá que ele não tem nenhuma classe nem id como atributos. Para acessar ele, já que é o elemento filho de uma div, usaremos o aninhamento de dois seletores CSS: "div h2". O único que precisamos fazer é usar o método .querySelector(), passar essa mesma string como argumento, e salvar o retorno numa variável

```
const segundoTitulo = document.querySelector("div h2");
console.log(segundoTitulo);
```



Finalmente, vamos acessar os mesmos elementos que acessamos no segundo exemplo da seção anterior (a div e o parágrafo com a classe "texto-simples") **para demonstrar duas diferenças chave** entre ambas abordagens. Usaremos o método `.querySelectorAll()` e passaremos como argumento a string `".texto-simples"`. Perceba que neste caso, como estamos usando um seletor CSS, **devemos preceder o nome da classe com um ponto**. Da mesma forma, se quisermos acessar um elemento via ID, o valor do id seria precedido por um símbolo de sustenido '#' (ex. "#titulo").

```
const textosPorClasse = document.querySelectorAll(".texto-simples");
console.log(textosPorClasse);
```

```
const textosPorClasse = document.querySelectorAll(".texto-simples");

console.log(textosPorClasse);
```

`getElementsByClassName` e `querySelectorAll()` retornam uma lista de elementos. `getElementById()` e `querySelector()` retornam apenas um único elemento.

Capturado elementos `li` numa variável `produtosSelecionados`, acessaremos o segundo elemento utilizando `produtosSelecionados[1]`

Muito bem! Quando capturamos elementos com os métodos `.getElementsByClassName()` e `.querySelectorAll()` acessamos eles da mesma forma que acessamos os elementos de um array. Sendo assim, o índice do primeiro elemento é 0, e o do segundo elemento é 1.

Usa-se o `querySelector()` e os seletores css para acessar os elemntos. usando-se `.` ou `#`

ao caturar uma lista pela `calss="usuário"` retorna `null` ao tentar imprimir a variável `usuarios`

```
document.getElementsByClassName('.usuario')
```

A string passada como argumento do `getElementsByClassName` deveria ser `usuario`. Correto! Quando usamos os métodos `.getElementsByClassName()` e `.getElementById()` passamos apenas o nome da `classe` e do `id` respectivamente. Já quando usamos os métodos `.querySelector()` e `.querySelectorAll()` precisamos colocar um ponto `'.'` antes das `classes`, e um `'#'` antes dos `IDs`.

5.5 - INNERTEXT E INNERHTML

5.5.1 - Innertext e Inner

`innerText` - Retorna o texto sem formatação e sem elementos HTML.

`innerHTML` - Retorna o texto com formatação e com elementos HTML.

5.5.1.1 - setup de arquivos

usaremos um projeto simples com dois arquivos: um `index.html` e outro `script.js`.

O arquivo `index` segue o padrão da estrutura base HTML, com os elementos dentro da tag: `<body></body>`

```
<body>

  <h1>Título da página</h1>
  <main>
    <h2>Subtítulo do conteúdo principal</h2>
    <section>
      <p class="paragrafo">
        Lorem ipsum dolor sit, amet consectetur adipisicing elit. Fugiat amet
        similique doloribus pariatur vel. Ullam, amet vero excepturi iusto,
        autem, dolore nihil perspiciatis ut aliquam sit laudantium voluptate
        perferendis possimus!
      </p>

      <p class="paragrafo">
        Lorem ipsum dolor, sit amet consectetur adipisicing elit. Dicta, qui
        hic consequuntur pariatur dolor alias, ut sed corporis laudantium
        quibusdam aliquid iusto tenetur officiis perspiciatis quasi,
        voluptates ipsum impedit recusandae. Lorem ipsum dolor sit amet
        consectetur adipisicing elit. Sequi minima, aliquam aut dolore
        consectetur voluptatum explicabo. Doloremque corporis veritatis nihil
        exercitationem velit distinctio deleniti voluptate, quis facilis ea
        consectetur fuga!
      </p>
    </section>
  </main>
</body>
```


Além disso, dentro da tag `<head></head>`, para conectá-la com o arquivo `script.js` que está na mesma pasta que o arquivo `index.html`, incluímos a tag:

```
<script src="script.js" defer ></script>
```

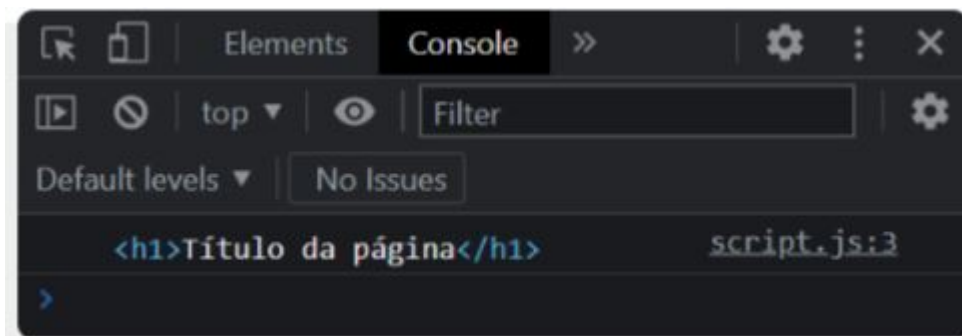
Por enquanto, o arquivo `script.js` está vazio e será nele que aprenderemos a usar as propriedades `innerText` e `innerHTML`.

5.5.2 - Acessando as propriedades INNERTEXT

Primeiro, abrimos nosso arquivo `script.js` e guardamos o elemento `<h1>` na variável chamada `elementoH1`, usando qualquer um dos quatro métodos aprendidos para acessar elementos do DOM.

```
let elementoH1 = document.querySelector("h1");  
console.log(elementoH1);
```

Observe que depois, imprimimos o `elementoH1` no terminal do navegador usando o método `console.log()`. Neste caso, usaremos o método `querySelector()`, passando a string `h1` como argumento.



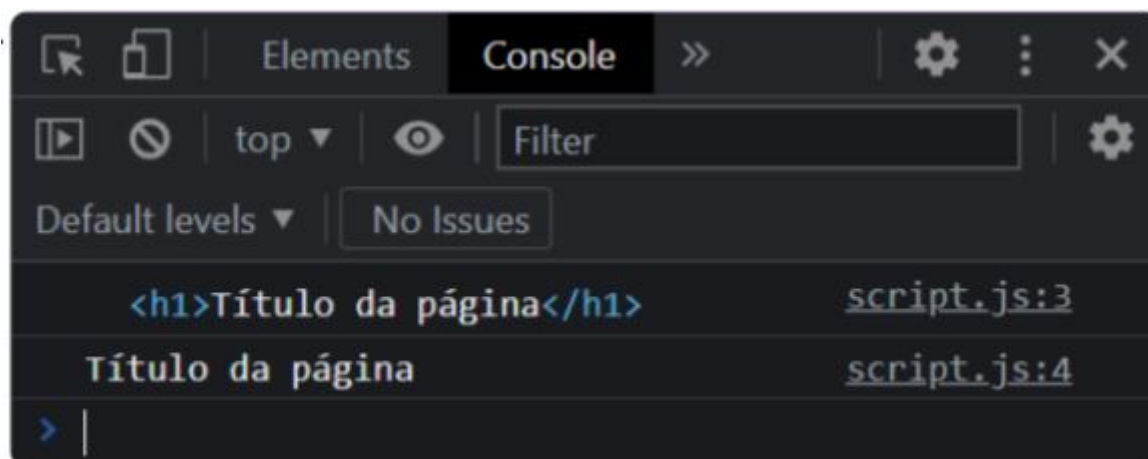
Resultado no console do navegador.

De volta ao arquivo `script.js`, usamos a função `dot notation` para acessar as propriedades, adicionando um ponto ao elemento capturado do DOM seguido do nome da propriedade (`innerText` ou `innerHTML`, respeitando as letras maiúsculas e minúsculas).

Depois, executamos mais um `console.log()` logo embaixo do primeiro, mas acessando a propriedade `.innerText`:

```
let elementoH1 = document.querySelector("h1");  
console.log(elementoH1);  
console.log(elementoH1.innerText);
```

Após salvar as mudanças e verificar o terminal no navegador, devemos ver o seguinte resultado:

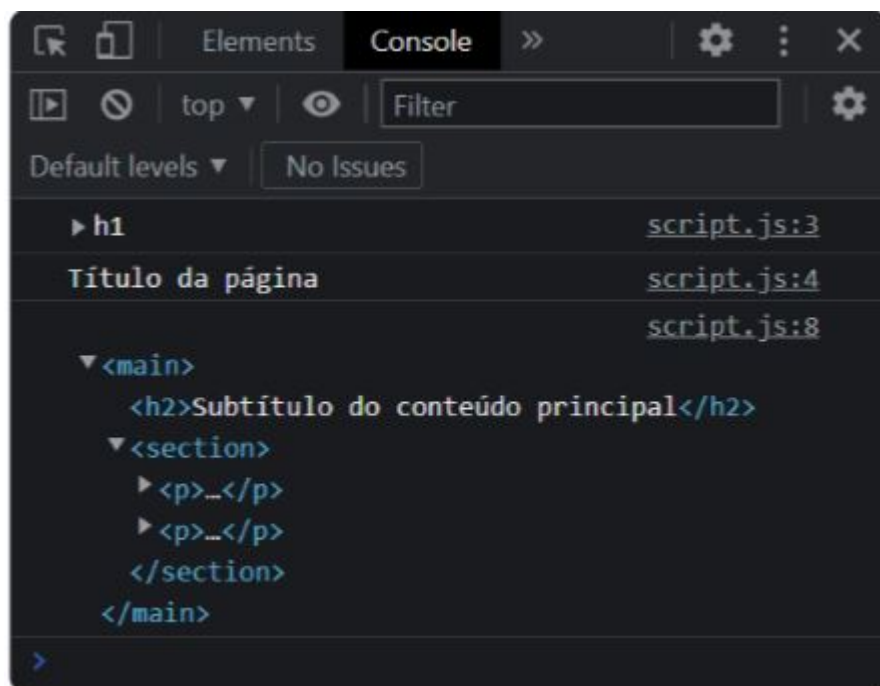


Na segunda linha, a propriedade `innerText` **retorna apenas o texto contido entre as tags** de abertura e de fechamento do elemento capturado do DOM.

5.5.3 - Acessando as propriedades InnerHTML

guardaremos o elemento `<main>` na variável `elementoMain`. Para isso, usamos o método `.querySelector` e o imprimimos no terminal com mais um `console.log()`

```
let elementoMain = document.querySelector("main");
console.log(elementoMain);
```

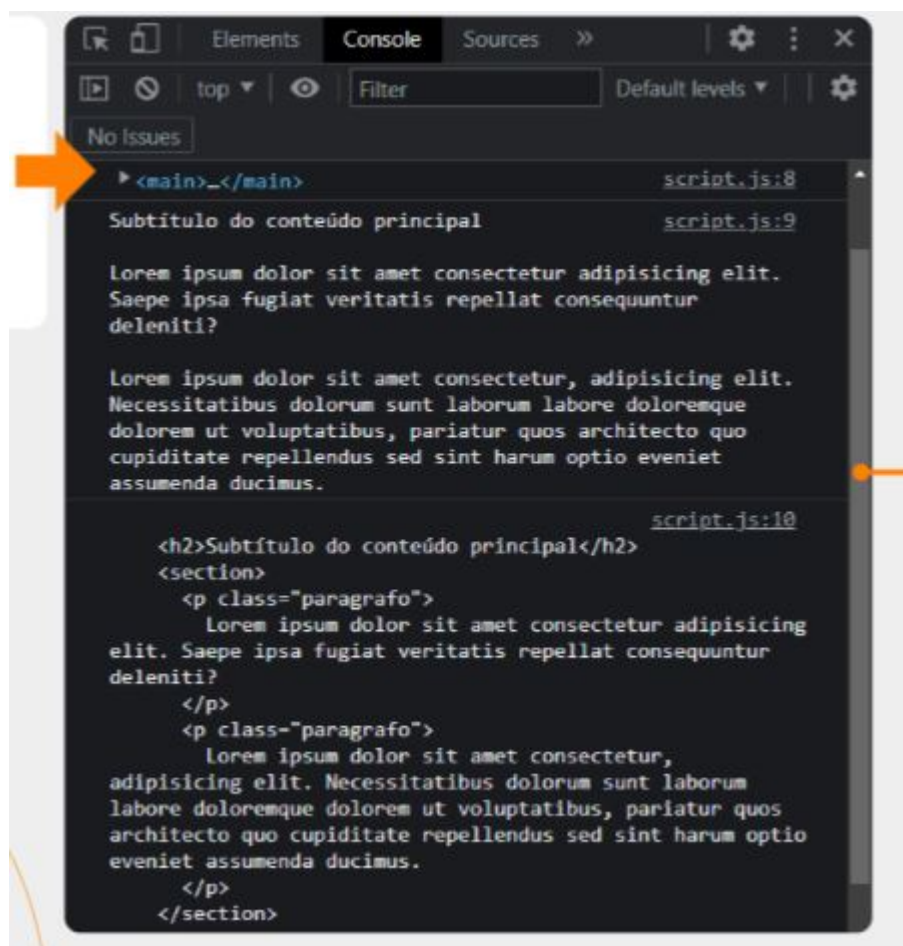


Resultado.

Clicando nas setas presentes ao lado esquerdo de cada tag de abertura, podemos expandir ou recolher seu conteúdo, sendo possível exibir os elementos filhos de cada elemento pai.

No arquivo `script.js`, vamos usar o `console.log()` mais duas vezes para imprimir nosso `elementoMain`, acessando as propriedades `.innerText` e `.innerHTML`:

```
let elementoMain = document.querySelector("main");
console.log(elementoMain);
console.log(elementoMain.innerText);
console.log(elementoMain.innerHTML);
```



resultado

Podemos ver que, mais uma vez, a propriedade `innerText` retornou apenas o texto contido entre as tags de abertura e de fechamento `<main> </main>`. Assim, mostra os textos dos elementos filhos, separando-os dos outros com uma quebra de linha.

Já a propriedade `innerHTML` retornou uma representação textual de todo o conteúdo HTML entre as tags de abertura e de fechamento do elemento `main`, mostrando os nomes de classe dos elementos `<p>`.

5.5.4 - Manipulando o DOM com `innerText`

Aparência da página:

Título da página

Subtítulo do conteúdo principal

Lorem ipsum dolor sit amet consectetur adipisicing elit. Saepe ipsa fugiat veritatis repellat consequuntur deleniti?

Lorem ipsum dolor sit amet consectetur, adipisicing elit. Necessitatibus dolorum sunt laborum labore doloremque dolorem ut voluptatibus, pariatur quos architecto quo cupiditate repellendus sed sint harum optio eveniet assumenda ducimus.

Podemos alterar, por exemplo, o texto do nosso elemento `h1` acessando a propriedade `innerText` do `elementoH1` e atribuindo a ele um novo valor com o operador de atribuição `=`, seguido do texto como uma string (escrito entre aspas simples ou duplas).

```
let elementoH1 = document.querySelector("h1");  
  
elementoH1.innerText = "Novo título com JS"
```

Resultado:

Novo título com JS

Subtítulo do conteúdo principal

Lorem ipsum dolor sit amet consectetur adipisicing elit. Saepe ipsa fugiat veritatis repellat consequuntur deleniti?

Lorem ipsum dolor sit amet consectetur, adipisicing elit. Necessitatibus dolorum sunt laborum labore doloremque dolorem ut voluptatibus, pariatur quos architecto quo cupiditate repellendus sed sint harum optio eveniet assumenda ducimus.

5.5.5 - Manipulando o DOM com innerHTML

Da mesma forma, podemos manipular o conteúdo HTML do nosso site usando a propriedade `innerHTML`. Nesse caso, substituiremos o conteúdo do `elementoMain` para conter um elemento `h2` com um texto diferente e uma lista não ordenada logo embaixo.

Para fazer isso, devemos acessar a propriedade `innerHTML` do `elementoMain` e atribuir uma string a ele contendo todo o conteúdo HTML desejado, de forma semelhante ao que escreveríamos no arquivo `index.html`.

```
let elementoMain = document.querySelector("main");  
  
elementoMain.innerHTML = `  
<h2>Novo subtítulo</h2>  
<ul>  
  <li>Elemento de lista JS 01</li>  
  <li>Elemento de lista JS 02</li>
```

```
<li>Elemento de lista JS 03</li>
</ul>
```

Novo título com JS

Novo subtítulo

- Elemento de lista JS 01
- Elemento de lista JS 02
- Elemento de lista JS 03

Resultado

5.5.6 - Conclusão

As propriedades `innerText` e `innerHTML` são simples e nos permitem manipular os elementos do DOM de forma muito ampla. Como orientação geral, **é recomendado usar a propriedade `innerText` quando queremos mudar apenas o texto de um elemento HTML que não possui elementos filhos**, pois, caso existam, eles serão substituídos pelo novo texto.

Já a propriedade `innerHTML` é **melhor ser usada quando queremos alterar o conteúdo HTML de qualquer elemento do DOM**, podendo incluir os elementos filhos, nomes de classe e qualquer outro atributo que os elementos HTML possam receber.

LEMBRE:se tag `<script>` que conecta o arquivo `script.js` com o arquivo `index.html` O atributo `defer`. Caso não colocarmos ele, o navegador provavelmente terá um erro de execução, e não mostrará nada na tela; Podemos usar qualquer um dos quatro métodos para capturar elementos do DOM sempre e quando chegarmos no resultado desejado; `innerText` e `innerHTML` **são propriedade** e não métodos; Com a propriedade `.innerHTML` é que, se usarmos aspas simples ou duplas, é muito difícil formatar o texto de uma forma que fique legível e intuitivo. Por esse motivo, **propomos usar dois acentos graves** como alternativa, pois dessa forma conseguimos quebrar linhas na nossa string e deixar ela mais parecida com a formatação usada quando escrevemos HTML. Esta abordagem é chamada de `template string` e é o próximo conteúdo que estudaremos!

5.5.7 - Exercício

1- Quando queremos manipular apenas um elemento simples sem elementos filhos e atributos, o que devemos usar? Isso mesmo! A propriedade `.innerText` é usada quando queremos mudar apenas o texto de um elemento HTML que não possui elementos filhos.

2 - Sérgio é programador e precisa atribuir um valor a um elemento do DOM, usando a propriedade `.innerHTML`. Qual símbolo ele deve usar para isso? Isso mesmo! Ao colocarmos o texto entre acentos graves, conhecidos pela utilização em crases, conseguimos quebrar a linha para organizar melhor o código.

3 - Como acessamos a propriedade `.innerText` de uma variável `nomeUsuario`, que guarda um elemento HTML do DOM? Isso mesmo! Para acessar a propriedade `.innerText`, basta chamar o nome da variável, seguida de

um ponto e o nome da propriedade.

5.5.8 - TEMPLATE STRING

5.5.8.1 - Template String

Quando trabalhamos com strings na linguagem JavaScript, é muito comum precisarmos concatenar strings, ou seja, juntar duas ou mais strings para gerar uma terceira. Contudo, com o aumento na complexidade das strings, surgiu a necessidade de um **recurso que permitisse trabalhar com elas de forma mais dinâmica**. Consequentemente, foram criados os **templates strings**.

```
let nome = "Rafael";
let sobrenome = "Pereira";

let nomeCompleto = "Meu nome é:" + nome + " " + sobrenome;
console.log(nomeCompleto);
```

Atribuímos duas variáveis com nome e sobrenome. Uma terceira variável foi declarada e recebendo a junção via "+" (concatenar).

5.5.8.2 - Interpolação na linguagem JavaScript

Concatenar strings usando o **método do exemplo anterior** *é uma solução simples* e rápida quando temos poucas variáveis e não precisamos personalizar a **string resultante**. o processo fica mais complexo e trabalhoso à medida em que as strings forem ficando maiores e precisamos incluir mais valores salvos em variáveis.

templates strings: Com eles, podemos ter **só uma string** e inserir os valores das novas variáveis nela. Esse processo é conhecido como **interpolação**. Para fazer isso, usamos dois acentos graves `` , usados para indicar crase, ao invés das aspas simples ou duplas.

Para inserir os valores das nossas variáveis, precisamos usar o símbolo cifrão \$, também conhecido como o símbolo do dólar, seguido de duas chaves {} com o nome da variável dentro delas.

```
${nome-da-variavel}
```

Vamos incluir os valores nome e sobrenome no nosso template string, usando essa estrutura e imprimindo a variável templateString com o método `console.log()`.

```
let nome = "Rafael";
let sobrenome = "Pereira";
```

```
let templateString = `Meu nome é: ${nome} ${sobrenome}`  
console.log(templateString);
```

VANTAGENS DA INTERPOLAÇÃO

1. escrever uma só string e inserir os valores desejados dentro dela;
2. As strings escritas com aspas **não permitem quebra de linha**, mas, com os *templates strings*, **podemos escrevê-las com quebras de linha sem causar erro** e elas serão interpretadas da mesma forma que escrevemos 1.

```
let templateString = `Meu nome é ${nome},  
meu sobrenome é ${sobrenome}`  
console.log(templateString)
```

Output:

Meu nome é Rafael,

meu sobrenome é Pereira

3. tudo que escrevemos dentro da estrutura `${ }` é interpretado pelo computador como JavaScript. Significa que podemos, por exemplo, **somar valores numéricos salvos em duas variáveis** e o *template string* exibirá o resultado da soma.

```
let numA = 5;  
let numB = 9;  
let soma = `A soma de ${numA} e ${numB} é ${numA + numB}`  
console.log(soma);  
  
//Esse exemplo imprimirá a frase A soma de 5 e 9 é 14.
```

5.5.8.3 - Exercício

1. Para inserir uma variável idade numa string usamos qual estrutura?
 1. Isso mesmo! Para inserir valores de variáveis nos templates strings, devemos colocar um cifrão seguido do nome da variável entre chaves `${idade}`.

2. Luciana é estudante do curso de Desenvolvimento de Sistemas. Para tirar boas notas em suas provas, ela deseja praticar a concatenação de strings na linguagem JavaScript. Para isso, qual é o site que ela deve usar?
 1. OneCompiler
3. Joana estuda programação, mas está achando complicado o conceito de templates strings. Para ajudá-la, você decidiu mostrar um dos benefícios de usar os templates strings. O que você diz a ela?
 1. Muito bem! Os templates strings nos permitem inserir qualquer tipo de expressão JavaScript dentro da estrutura ``{}``. Inclusive, com eles, podemos adicionar a soma de valores diretamente ou salvá-los em variáveis.

5.5.9 - CRIANDO ELEMENTOS NO DOM

5.5.9.1 - Criar elementos no DOM

Na aula anterior aprendemos a alterar/incluir conteúdo, porém muitas vezes não será esse nosso objetivo. Às vezes, apenas precisamos adicionar um conteúdo novo, sem apagar ou manipular o conteúdo original do elemento.

Há várias formas, mas focaremos em duas `createElement` e `appendChild`.

5.5.9.1.1 - setup de arquivos

index.html java.js

conteúdo abaixo dentro da tag `body`

```
<body>
  <ul class="lista-linguagens">
    <li class="ling-html">HTML</li>
    <li class="ling-css">CSS</li>
  </ul>

  <section class="postagens">
    <div #="post-html">
      <h2 class="post-titulo">HTML</h2>
      <p class="post-texto">
        HTML é uma linguagem de marcação
      </p>
    </div>
    <!-- fim #post-html -->
    <div id="post-css">
      <h2 class="post-titulo">CSS</h2>
      <p class="post-text">
        CSS é uma linguagem de estilização
      </p>
    </div>
    <!-- fim #post-css -->
  </section>
  <!-- fim section .postagens -->
```



```
</body>
```

5.5.9.2 - Criar elementos SIMPLES no DOM

Para adicionar um elemento à lista não ordenada, seguiremos três etapas:

- criar um elemento HTML;
- popular e manipular esse elemento HTML;
- adicionar o elemento HTML no DOM.

PRIMEIRA ETAPA - CRIAR ELEMENTO

Vamos criar um `` e salvá-lo em uma variável para facilitar sua manipulação na próxima etapa. Iniciaremos chamando o objeto “`Document`” e usando o método dele `.createElement()`. Esse método recebe como argumento uma string com o **nome do elemento que queremos criar**. O nome do elemento será o mesmo que usaremos para abrir e fechar as tags dos elementos HTML. Assim, passamos como argumento o valor “`li`”:

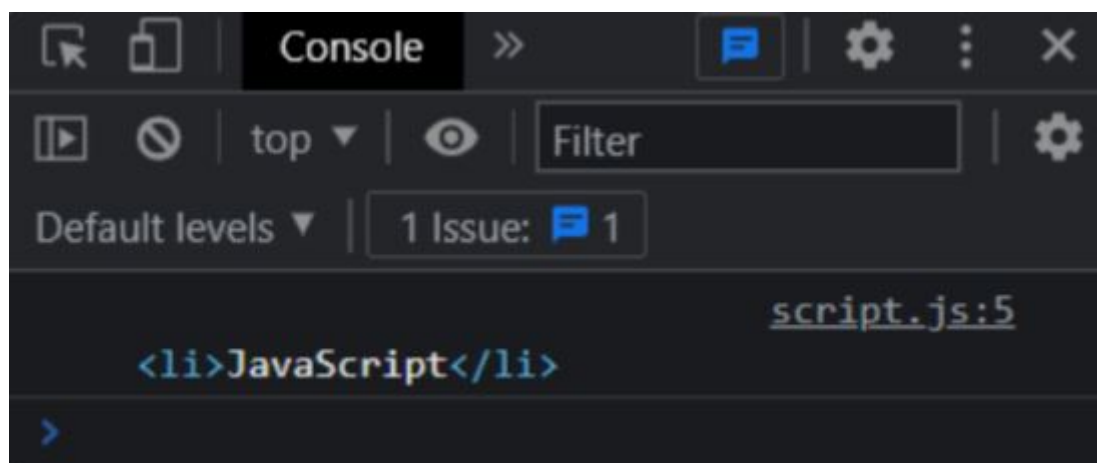
```
let elementoJavaScript = document.createElement("li");

console.log(elementoJavaScript);
//mostra no console as "li" criadas
```

Iniciaremos usando a propriedade `innerText` para adicionar o texto “JavaScript” no elemento criado `li` ele.

```
let elementoJavaScript = document.createElement("li");
elementoJavaScript.innerText = "JavaScript"

console.log(elementoJavaScript);
```



SEGUNDA ETAPA - POPULAR/MANIPULAR ELEMENTO

Além do `innerText`, podemos usar outras propriedades para manipular os elementos HTML que criamos. Nesse caso, sabemos que os elementos da **nostra lista tem um id** que os diferencia. Dessa forma, *podemos adicionar um id ao nosso elemento*, usando a propriedade `.id` do `elementoJavaScript`, e atribuir a ele um valor. Observe a imagem.

TERCEIRA ETAPA - ADICIONAR NO DOM

Com o elemento **criado, populado[^1] e manipulado**, podemos adicioná-lo ao nosso site.

[^1]: No contexto de informática, "popular" é um verbo que significa "preencher com conteúdo", "povoar". É uma tradução bruta do inglês "populate", que acabou por se implantar no dialeto da informática

Para fazer isso, **precisamos capturar o seu elemento pai** via DOM e salvá-lo em uma variável.

Nesse caso, o elemento pai é a lista não ordenada com a classe `Lista-linguagens`. Sabendo disso, usaremos o método `.querySelector` para capturá-lo e guardá-lo dentro de uma variável.

```
let elementoJavaScript = document.createElement("li");
elementoJavaScript.innerText = "JavaScript"
elementoJavaScript.id = "ling-js"

let listaLinguagens = document.querySelector(".lista-linguagens");
```

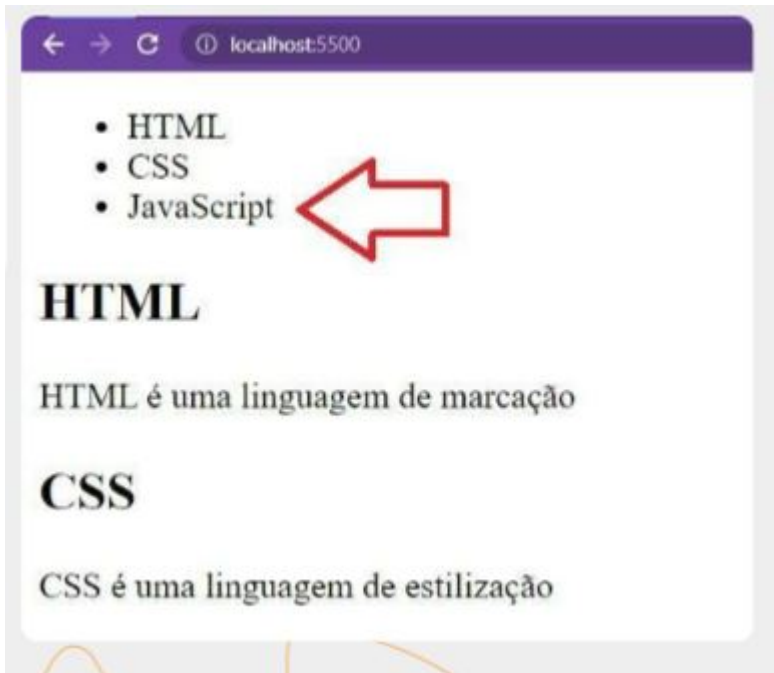
Com o elemento da lista não ordenada salvo na variável `listaLinguagens`, podemos chamar essa variável e usar o método `appendChild()` para adicionar elementos nele.

Para fazer isso, *basta passar o que deseja como argumento dentro dos parênteses*.

No nosso exemplo, vamos adicionar o `elementoJavaScript` que criamos e manipulamos nas etapas anteriores. Observe a imagem.

```
let elementoJavaScript = document.createElement("li");
elementoJavaScript.innerText = "JavaScript"
elementoJavaScript.id = "ling-js"

let listaLinguagens = document.querySelector(".lista-linguagens");
listaLinguagens.appendChild(elementoJavaScript);
```



OBSERVAÇÃO IMPORTANTE: é que a inserção dinâmica de conteúdos no nosso site, usando JavaScript, **não altera o código fonte** dos nossos arquivos HTML. Se você verificar o arquivo `index.html`, verá que ele não foi alterado.

5.5.9.3 - Criar elementos COMPLEXOS no DOM

O exemplo anterior é **especialmente útil** quando queremos criar e adicionar **elementos simples**, ou seja, *que não possuem elementos filhos e que têm poucos ou nenhum atributo*

adicionaremos uma nova postagem ao elemento `<section>`.

Se você conferir o arquivo `index.html`, verá que cada postagem é um elemento `<div>`, com um `id` e dois elementos filhos. Além disso, cada elemento filho tem seu respectivo texto e classe.

Para não precisar criar três elementos diferentes (`div`, `h2` e `p`) e manipular cada um deles adicionando textos e classes, criaremos apenas um elemento e usaremos a propriedade `innerHTML`, seguindo as mesmas três etapas do exemplo anterior.

PRIMEIRA ETAPA

Criamos o elemento `<div>` para a postagem e guardamos ele na variável `postagemJavaScript`:

```
const postagemJavaScript = document.createElement("div");
```

SEGUNDA ETAPA

Usamos a propriedade `innerHTML` para inserir todo o conteúdo HTML das postagens em um template string:

```
const postagemJavaScript = document.createElement("div");

postagemJavaScript.innerHTML =
```

```
`<h2 class="post-titulo">JavaScript</h2>

<p class="post-texto">

  JavaScript é uma linguagem de programação

</p>`
```

Lembrando que usar o template string nessa etapa nos permite adicionar qualquer outro conteúdo salvo em uma variável. Para isso, o recurso `${ }` é usado.

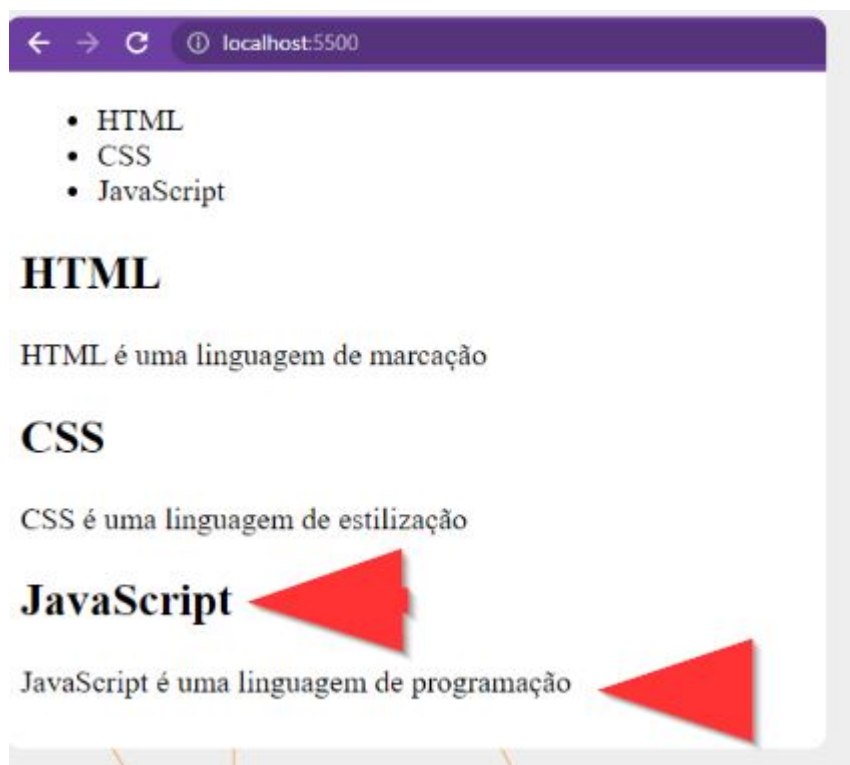
Nessa etapa, capturamos o elemento pai da nossa postagem e salvamos ele em uma variável:

```
const postagens = document.querySelector(".postagens");
```

Nele, adicionamos o elemento postagemJavaScript através do método `appendChild()`.

```
const postagens = document.querySelector(".postagens");
postagens.appendChild(postagemJavaScript);
```

Resultado:



Conclusão:

Os métodos `createElement()` e `appendChild()` nos permitem criar elementos HTML no DOM e inseri-los no nosso site de forma dinâmica.

Usando eles em conjunto com os conceitos aprendidos nas aulas anteriores (`innerText`, `innerHTML` e template strings), **conseguimos manipular os elementos preexistentes no site e populá-lo com novas informações usando a linguagem JavaScript.**

Util em duas situações:

1. quando queremos popular nosso site com dados vindo de sistemas externos;
2. quando queremos que a interação de nossos usuários altere o conteúdo do site;

Referências bibliográficas

[DEVFURIA. JavaScript - Create Element](#)

[SOFTAUTHOR. append\(\) vs appendChild JavaScript](#)

[MATHEUS BATTISTI - HORA DE CODAR. Curso JavaScript #38 - Criando elementos com DOM \(createElement\).](#)

Exercício

1. Qual é o método usado para adicionar o elemento produto ao elemento carrinho?
 1. `carinho.appendChild(produto)`. Permite anexar um elemento a seu respectivo elemento pai.
2. Qual a forma correta de criar um elemento section no DOM?
 1. `document.createElement("section");`
3. Você e Pedro são colegas no curso de programação e, ao estudar, ele afirma que a ordem das etapas para adicionar um elemento no DOM é: (1) criar o elemento HTML, (2) popular ou manipular o elemento criado e (3) adicionar o elemento em um arquivo .html. Pedro está correto?
 1. Não, pois a terceira etapa é adicionar o elemento no DOM. Isso mesmo! Os elementos que criamos não são adicionados em um arquivo .html original, eles são adicionados ao DOM e interpretados pelo navegador.

[Link material CodePark07 exercício](#)

Comentários ao CodePark07 Vale destacar que, *embora não seja mencionado um layout específico nas orientações, a ordem na qual adicionamos nossos elementos no DOM pode ser relevante em casos como o apresentado*. Já que ambos os elementos criados ('`título`' e '`produto`') estão sendo adicionados no mesmo "elemento pai" (neste caso, o elemento '`body`'), se usarmos o método `.appendChild()` primeiro passando a variável '`produto`' como argumento, e depois com a variável '`título`', teremos na nossa página primeiro o produto e depois o título do site. Novamente, isto **é só um detalhe que precisamos levar em consideração** quando adicionamos **mais de um "elemento filho" no mesmo "elemento pai"**, já que o navegador lerá nosso arquivo JavaScript de cima para baixo e executará cada método na ordem que for lendo eles.

5.6 REVISÃO JAVASCRIPT I

5.6.1 - Variáveis

Para declarar variáveis, usamos quatro elementos:

- a palavra reservada para o tipo da variável;
- o nome da variável;
- o operador de atribuição "=";
- o valor a ser atribuído à variável. Exemplo: `var letra = "a"`

Temos 3 tipos de variáveis: `var` : caiu em desuso; `let`: variáveis que podem ter novos valores atribuídos
`const`: variáveis que não podem ter seu valor reatribuído.

5.6.2 - Operadores

Aritméticos e relacionais são similares com Python;

Lógicos são diferentes, sendo:

OPERAÇÃO	OPERADOR
Conjunção	&&
Disjunção	
Negação	!

5.6.3 - Estruturas Condicionais

3 elementos são necessários:

1. ao menos uma palavra reservada, como `if` e `else`
2. uma condição entre parênteses que retorna um valor booleano;
3. um bloco de código entre chaves `{}`;

```
if(num >5){  
    console.log("é maior que 5")  
} else {  
    console.log("É menor que 5")  
}
```

É utilizada `else` após o `if` para ser executada um bloco caso a condição do `if` seja falsa.

Podemos encadear mais de uma verificação utilizando entre o `if` inicial e o `else` final os `else if`.

```
if(num >5){  
    console.log("é maior que 5")  
} else if (num <5) {  
    console.log("É menor que 5")  
} else {  
    console.log("O número é 5")  
}
```

Aprendemos também que a linguagem JavaScript **nos permite verificar a veracidade de um valor booleano**.

Para isso, basta escrever o nome da variável que guarda o valor.

```
if(aprovado == true)...  
//é o mesmo que  
if(aprovado)...
```

Podemos verificar se o valor é falso utilizando o operador de negação a frente do nome da variável.

```
if(aprovado == false)...  
//é o mesmo que  
if(!aprovado)...
```

5.6.4 - Loops

A estrutura base de um **for loop** consiste em :

- uma palavra reservada **for**;
- parâmetros **loop** entre parênteses **()**;
- um código, que será executado, entre chaves **{}**;

Dentro dos parênteses, escrevemos, separados por ponto e vírgula, os três parâmetros do **loop**:

1. variável contadora;
2. condição de parada;
3. incremento (ou decremento);

Exemplo:

```
for(let i = 0; i < 5; i++){  
  console.log(i);  
}
```

- temos um **for loop** que começa com uma variável contadora **i** de valor inicial **0**;
- incremento de **1** a cada repetição;
- a condição de parada é o contador continuar a ser **menor que 5**. Ser igual ou maior cancelará o loop.

5.6.5 - Arrays

Podemos percorrer arrays usando o **for loop** e a propriedade **.length()**, que retorna o tamanho de um array para compor a condição de parada. Observe o exemplo:

```
let queijos = ['mussarela', 'prato', 'brie'];

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

5.6.5 - Funções

Na linguagem JavaScript, temos **três formas** de declarar variáveis:

1. declaração regular;

1. Para declarar funções na forma regular, precisamos de quatro elementos: a palavra reservada `function`, o nome da função, os parâmetros que a função pode receber entre parênteses e o bloco de código a ser executado entre chaves.

```
function subtrair(a,b){
  return a - b
}
```

2. declaração anônima;

1. Já as funções anônimas **são declaradas sem um nome**, apenas com a palavra reservada `function`, os parâmetros entre parênteses e o bloco de código entre chaves. Vale lembrar que, **este código geralmente é atribuído a uma variável** que guarda a função como o seu valor.

```
const subtrair = function(a,b) {
  return a-b
}
```

3. arrow functions;

1. A sintaxe das *arrow functions* nos permite **simplificar a escrita das funções anônimas**. Ela é extremamente **útil** para as **funções pequenas**, com um ou nenhum parâmetro, e os blocos de códigos pequenos. Observe o exemplo:

```
const numAnterior = num => num-1
```

5.6.6 - JavaScript e HTML

Aprendemos duas formas de inserir um código JavaScript nos nossos arquivos HTML: por meio de **tags** ou de **arquivos.js**.

A tag `<script></script>` nos permite escrever o código JavaScript diretamente nos nossos arquivos HTML.

Caso usemos a função `console.log()` para imprimir algo, isto será exibido no terminal do navegador. Para conferir o terminal do navegador, abrimos as **Dev Tools** do navegador e selecionamos a aba **Console**.

Além disso, entre as tags de abertura e de fechamento do script, podemos escrever qualquer tipo de código JavaScript.

5.6.7 - Arquivo .js

Aprendemos que uma boa prática é escrever nossos códigos JavaScript em um arquivo separado com a extensão .js e conectá-lo com seu arquivo HTML.

Para conectar `arquivos.js`, também podemos usar a tag `<script>` no arquivo HTML.

Nesse caso, **não escrevemos no espaço entre as tags de abertura e de fechamento**. No entanto, adicionamos dois atributos à tag de abertura: `src` para definir a rota até o arquivo.js e `defer` para indicar ao navegador que deve executar nosso código JavaScript apenas depois que todos os elementos HTML

```
<script src="./script.js" defer></script>
```

5.6.7 - DOM

Aprendemos o que é o Document Object Model (DOM) e como podemos acessar nossos elementos HTML usando-o em conjunto com a linguagem JavaScript.

Vamos relembrar que o DOM é um modelo que representa todos os elementos exibidos em uma página web.

5.6.7 - Acessando DOM por #id e .classe

O DOM nos fornece o objeto **“Document”** para acessar os elementos HTML do nosso site, usando diferentes métodos, que são as funções salvas como valores de um objeto e acessadas via **dot notation**.

O método `.getElementById()` recebe uma string como parâmetro, que **deve ser igual** ao valor do `id` do elemento que queremos acessar. O retorno dessa ação é, geralmente, guardado em uma variável. Veja o exemplo:

```
let titulo = document.getElementById("titulo");
```

O método `.getElementsByClassName()` recebe, como parâmetro, uma string que deve ser igual ao nome da `classe` dos elementos que queremos acessar.

```
let postagens = document.getElemenByClassName("Postagens");
```

5.6.7 - Acessando DOM com seletores CSS

Podemos acessar elementos HTML usando os mesmos seletores que usamos quando trabalhamos com CSS.

Para acessar **um único elemento**, o método recomendado é o `.querySelector()`. Ele recebe uma string como parâmetro com o *seletor*, ou com uma *mistura de seletores*. O código deve ser escrito da mesma forma que faríamos em um arquivo CSS, ou seja, com os seletores de classe começando com ponto “.”, os seletores de id começando com símbolo de cerquilha “#”, que é como “jogo da velha”, etc. Veja o exemplo:

```
let tituloPostagens = document.querySelector(".lista-postagens h2");
```

Para acessar vários elementos de uma só vez com seletores CSS, o método recomendado é o `.querySelectorAll()`. Ele recebe uma string como parâmetro com o seletor, ou uma mistura de seletores, e é escrito da mesma forma que o escreveríamos em um arquivo CSS. Veja o exemplo:

```
let tituloPostagens = document.querySelectorAll("#postagens > . div");
```

5.6.8 - innerText e innerHTML - Acessando informação

Usamos as propriedades `.innerText` e `.innerHTML` para acessar os conteúdos de texto e em HTML dos nossos elementos **capturados** via DOM.

A propriedade `.innerText` **retorna** *todo o texto contido entre as tags de abertura e de fechamento do elemento capturado* via DOM. Assim:

```
let titulo = document.getElementById("titulo");  
console.log(titulo.innerText)
```

Caso o elemento acessado possua elementos filhos, a propriedade retornará o texto de ambos, que estarão separados por uma quebra de linha.

Já a propriedade `.innerHTML` **retorna** *todo o conteúdo HTML contido entre as tags de abertura e de fechamento do elemento capturado via DOM*, incluindo os elementos filhos, seus textos e seus atributos, mas sem a quebra de linha. Veja o exemplo:

```
let linksDeNavegacao = document.querySelectorAll("nav ul");  
console.log("linksDeNavegacao.innerHTML")
```

5.6.9 - innerText e innerHTML - Manipulando informação

Além de acessar os conteúdos textuais e os de HTML usando as propriedades `.innerText` e `.innerHTML`, **podemos atribuir novos valores** a eles. Para isso, **usamos o operador de atribuição “=”** e uma **string** com o novo valor. Veja o exemplo:

```
titulo.innerText = "Titulo inserido com JavaScript"
```

Para organizar a escrita das strings atribuídas como valores à propriedade `.innerHTML`, recomendamos usar os *templates strings*. Assim:

```
linksDeNavegacao.innerHTML = `  
<li> Elemento inserido com JavaScript </li>  
`
```

5.6.10 - Templates String

Os templates strings possuem vantagens em relação à concatenação de strings.

Na linguagem JavaScript, podemos concatenar, ou juntar, duas ou mais strings se usarmos o operador de concatenação `“+”`.

As strings agrupadas podem ser escritas normalmente ou usando apenas o nome da variável que guarda a string. Veja o exemplo:

```
let nome = "Fulano";  
let sobrenome = "Perez";  
  
let boasVindas = "Olá" + nome + " " +sobrenome
```

Porém, uma **alternativa mais prática** para as strings concatenadas, ou as mais complexas, é o **template string**. Nele, escrevemos uma única string entre dois acentos graves (`` ``), usados na crase, e **inserimos qualquer código JavaScript usando a estrutura `${ }`**. Observe o exemplo:

```
let nome = "Fulano";  
let sobrenome = "Perez";  
  
let boasVindas = `Olá  ${nome}  ${sobrenome}`
```

Além disso, os templates strings nos permitem escrever uma string com quebras de linha. Observe o exemplo:

```
let variasLinhas = `Primeira linha.  
Segunda Linha.  
E todo isto é ainda uma string só`
```

5.6.10 - Elementos HTML com código JavaScript

Por fim, aprendemos a criar elementos HTML com código JavaScript e a adicioná-los de forma dinâmica ao nosso site usando o DOM.

Para realizar essas ações, precisamos seguir três etapas:

- criar o elemento HTML;
- popular ou manipulá-lo;
- adicioná-lo a um elemento pai;

Para criar um elemento, chamamos o objeto “**Document**” e usamos o método `createElement()`. Depois, passamos como argumento uma string com o nome do elemento HTML desejado.

Geralmente, o retorno desse método é guardado em uma variável. Veja o exemplo:

```
let novoSubtitulo = document.createElement("h3");
```

Para **popular** o elemento criado, podemos usar as propriedades `.innerText` ou `.innerHTML`, dependendo da complexidade do conteúdo. Observe:

```
novoSubtitulo.innerText = "Comentários dos clientes"
```

Além disso, podemos adicionar **atributos** ao elemento seguindo o mesmo processo. Dessa forma:

```
novoSubtitulo.id = "comentarios"
```

Para **adicionar o elemento criado à nossa página**, devemos:

- capturar o elemento pai, que o contém via o DOM;
- guardar o elemento criado em uma variável;
- chamar o método `appendChild()` no elemento pai, passando como argumento a variável que guarda o elemento criado. Veja o exemplo:

```
let secoes = querySelector(".secoes");  
secoes.appendChild(novoSubtitulo)
```

Vale lembrar que a inserção de conteúdos textuais ou HTML que usam a linguagem JavaScript **não altera nossos arquivos HTML**.

As mudanças implantadas são **apenas refletidas no navegador que interpreta o código do nosso site**.

Referências Bibliográficas:

[DEV APRENDER. Curso Javascript Completo 2020 \[Iniciantes\] + 14 Mini-Projetos. 14 mar. 2020. Disponível em:](#)

[FLÁVIO COUTINHO. \[JS4 - tópico 2\] Criando elementos HTML dinamicamente com JavaScript. 24 ago. 2019](#)