Javascript

Declaração de variável.

`let` palavra reservada para declarar variavel.

let variable be something…

```
let variavel = 3
```

`let` permite que a variável seja mutável no futuro.

Com essa variável é possível fazer qualquer operação com varíaveis. A sintaxe é bem similar ao python.

Declaração de função:

`function` palavra reservada para criar funções.

```
function funcao() {
  logica
}
```

Interação com HTML.

Primeiro, é preciso que o código HTML tenha uma **tag de script** que execute o arquivo .js que estamos escrevendo.

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

Dessa forma, o arquivo JS será executado no site.

Para mudar o conteúdo de divs no pelo javascript, as divs precisam de um `ID` para que o javascript consiga localizar o objeto da div (parecido com o webelement) e alterá-lo dentro do script.

Para pegar um objeto div no javascript e armazená-lo numa variável, utilizaremos a função do objeto **document (é o objeto da página)** `document.` `getElementById` `(id da div)`

```
let div = document.getElementById("id")
```

Agora, pode-se mudar os atributos de div como innerText para alterar o seu valor no código html.

Se o texto de uma div estiver vazia e você mudá-lo, o texto irá aparecer.

DOM - Document Object Model

O javascript recebe a página HTML como um objeto chamado `document`

Esse objeto possui diferentes métodos e atributos para scrapar os dados do html e alterá-los.

Essa é a forma que o JS tem de alterar o código HTML.

Outro tipo de get

`document.querySelector(prefixo + nome do selecto )`

Isso retornará a tag que tenha o prefixo + o nome do selector informado.

Prefixo:

- # → id
- . → Classe css

Nada → Tags nativas do html

If e Else

Sintaxe do if (igual ao do c e c++)

```
if (expressao) {
}

else if (expressao 2) {
}

else{
}
```

Variaveis Booleanas

True: `true`

False: `false`

Arrays

`let lista = []`

A lista é declarada igual ao python.

O JS tem indexação que começa em 0.

Array é um objeto, então ele tem atributos.

Podem ter elementos mistos nela.

Atributos da lista.

lista.**lenght**: retorna o tamanho da lista. A quantidade de listas

Métodos da lista

lista.**push**(item): adiciona o item na lista.

lista.pop(): remove o último item da lista.

For Loops:

Sintaxe igual a do c++

```
for (declara variavel; condicao de parada; i++) {

}

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

Modulo math.

Math.random()

Math.ceil()

Math.floor()

Incrementador I++ e Decrementador I—

Objetos

Objetos em JS são como dicionários em python.

```
let objeto = {
  atributo1: 10,
  atributo2: "oi"
}
```

Os atributos são consultados com objeto.atributo ou com objeto[”atributo”].

Métodos

```
let objeto = {
  atributo1: 10,
  atributo2: "oi"
  metodo1: function(){
    Sintaxe da função
    return
  }
}
```

São definidas dentro das chaves.

Rocketseat

Arrow Function:

São formas de criar funções anônimas e rápidas.

São úteis para entregar um retorno ao código que precisa ser processado por uma função sem ter que criá-la usando `function`

Como criar:

```
let variavel = (a, b) => a+b
```

Nesse caso, variável vira uma função com os parâmetros (a, b)

Existe mudança de escopo do this.

Callback function

É a chamada (call) de uma função dentro de outra, quando essa primeira função é passada como um parâmetro da outra função.

__Proto__

```
objeto.__proto
```

O fato de que o javascript já criou vários protótipos dos tipos e estruturas de dados que a gente usa, como listas, dicionarios, int etc.

Isso significa que eles já tem metodos pré-produzidos. Facilitando o uso. Como string.toLowerCase()

Casting

Variavel para numero: `Number(variavel)`

Variavel para string: `String(variavel)`

Delimitar Casas Decimais

```
float .toFixed(casas decimais)
```

Criar array a partir de strings ou numeros.

```
let lista de caracteres = Array.from(string ou numero)
```

Manipulação de array.

array.push(elemento) → insere no fim

array.pop() → apaga o ultimo elemento

array.unshift(elemento) → insere no inicio

array.shift() → remove do inicio

array.sllice(inicio, fim) → slice do array

array.splice(x, y) → remove os elementos de indice, x, y

array.indexOf(elemento) → retorna o indice do elemento.

New

`new` é uma palavra reservada que chama o construtor de uma classe. É minúsculo

Delete objeto.atributo

Deleta o atributo do objeto.

Switch

Igual ao c ou c++

Throw

palavra reservada que joga um erro na expressao

`throw “deu erro”` → Ele retornará o deu erro. Ela para a execução da função.

Try e catch

Try é igual ao python

O catch é o exception, mas ele é uma função que se passa uma variavel como parametro para guardar o erro nela.

```
function funcao(nome){
  if (nome === "banana")
    throw "erro"
    // para a execução
}


try {
  let nome = "banana"
```

```
  funcao(nome)
// da erro por conta do nome
}

catch(erro){
  console.log(erro)
}
```

For of e For in

For of pega os elementos de uma lista.

`for (item of lista) console.log(item)`

For in pega os atributos de um objeto.

`for (atributo in objeto) console.log(atributo)`

F-string em JS

em uma string, usamos o simbolo “${variavel}” para colocar a variavel dentro da string.

DOM → Document Object Model.

É a página HTML em forma de objeto.

`document` é o objeto html, e possui diversos métodos para manipulação e consulta dos itens do html.

`document.getElementById()`

Pega o elemento do html pelo seu id.

`document.getElementByClassName()`

Pega o elemento do html pelo seu id.

`document.getElementByTagName()`

Pega o elemento do html pela sua tag.

`document.querySelector(Seletor)`

Pega o elemento do html pelo seu seletor, é mais geral que os outros.

Talvez mais lento que o getElementById()

`document.querySelectorAll(Seletor)`

Pega todos os elemento do html pelo seu seletor.

Manipulação de conteúdo.

Ao capturar um elemento, podemos mudar os seus atributos e conteudos.

elemento.textContent → Texto do elemento.

elemento.innerText → Texto do elemento.

elemento.innerHTML → Colocar conteúdo html como filho de um elemento.

elemento.value → Valor de um input.

elemento.classList → Retorna uma lista de classes que estão sendo usadas pela tag.

Pode-se adicionar, remover ou alternar a ativação de qualquer classe desejada com os métodos.

- **elemento.classList.add(”classe”)**
- **elemento.classList.remove(”classe”)**
- **elemento.classList.toggle(”classe”)**

Usado para mudar o estilo da tag.

`.setAttribute(”atributo”, “valor”)` → Método para setar o valor de um parametro de uma tag.

`.removeAttribute(”atributo”)` → Método para remover o atributo x

`elemento.style.propriedade` = “x” → Muda o estilo do elemento baseado na propriedade

Navegação pelos itens pais e filhos.

`elemento.parentNode` → Nó pai

`elemento.parentElement` → Elemento Pai.

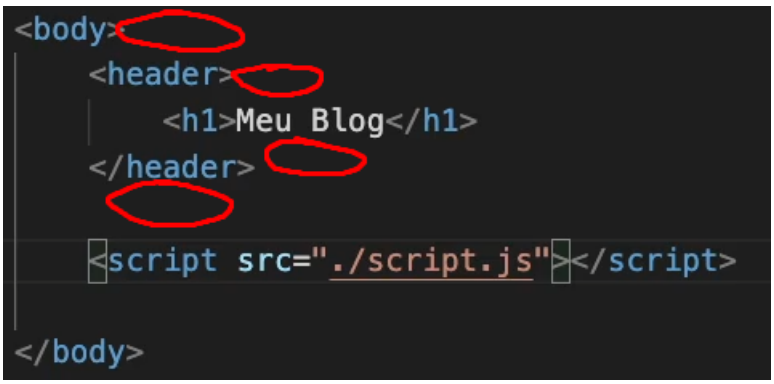
`elemento.childNodes` → Lista de filhos

`elemento.children` → HTML Collection

`elemento.firstChild` → Primeiro elemento considerando o espaço entre as tags, se tiver espaço, pegará o espaço.

`elemento.firstElementChild` → Primeiro elemento desconsiderando os espaços entre as tags.

obs: Essas tags com Element no meio, desconsidera o espaço entre as tags.



Exemplos de espaços vazios.

`elemento.lastChild`

`elemento.lastElementChild`

`elemento.nextSibling / nextElementSibling`

`elemento.previousSibling /previousElementSibling.`

Criar elementos no JS.

`let elemento = document.createElement("tag").`

Esse elemento fica salvo na memória do JS e pode sofrer alterações. **Mas ele ainda não foi implementado no código HTML da página.**

Para implementá-lo, precisamos **encontrar um ponto de partida**: fazer um query em um elemento.

`const body = document.querySelector("body").`

Para adicionar na frente desse ponto → `body.append(elemento).`

Para adicionar atrás desse ponto → `body.prepend(elemento).`

E se eu quiser inserir em uma posição específica?

Se eu quero inserir um elemento em uma posicão específica, eu preciso pegar o elemento que está na frente dessa posição e o seu pai.

Assim consigo usar o método pai**.insertBefore**(elemento a inserir, elemento da frente)

```
<body>
  <h1> Oi! </h1>
  -------- Aqui ----------
  <script src="index.js">
  </script>
</body>
```

Se eu quiser inserir alí, eu teria que pegar o pai BODY, o elemento SCRIPT e o elemento que quero inserir.

Aí ficaria: `body.insertBefore(elemento, script)`

Não existe um insert after, então se eu quiser inserir algo depois de um elemento, no caso, depois do sibling.

Eventos

As tags html têm um atributo que começa com **on,** geralmente são onclick, ondrag, ondblclick (double click) e a gente iguala eles à uma função do javascript.

Exemplo:

`<h1 onclick=”funcao()”>Oi</h1>`

Outra forma de adicionar eventos

Eventos de teclado.

Dado um input no html, podemos fazer um query dele no JS e guardar num elemento.

`const input = query(input)`

Agora, input tem metodos pra esperar eventos.

`input.onkeypress() = function()`

Executa a função ao pressionar um botao

EventListeners

const elemento = query(elemento)

elemento.addEventListener(”tipo do evento”, funcao()) → Melhor forma de se configurar eventos.

Classes

Criar classe

```
class Nome {
  constructor() {
    this.atributo = "tal atributo";
  }

  metodo() = {
    funcao;
  }
}
```

Herança

```
class filho extends classe_pai {
  constructor() {
    super(); //Puxa os atributos do pai.
  }
}
```

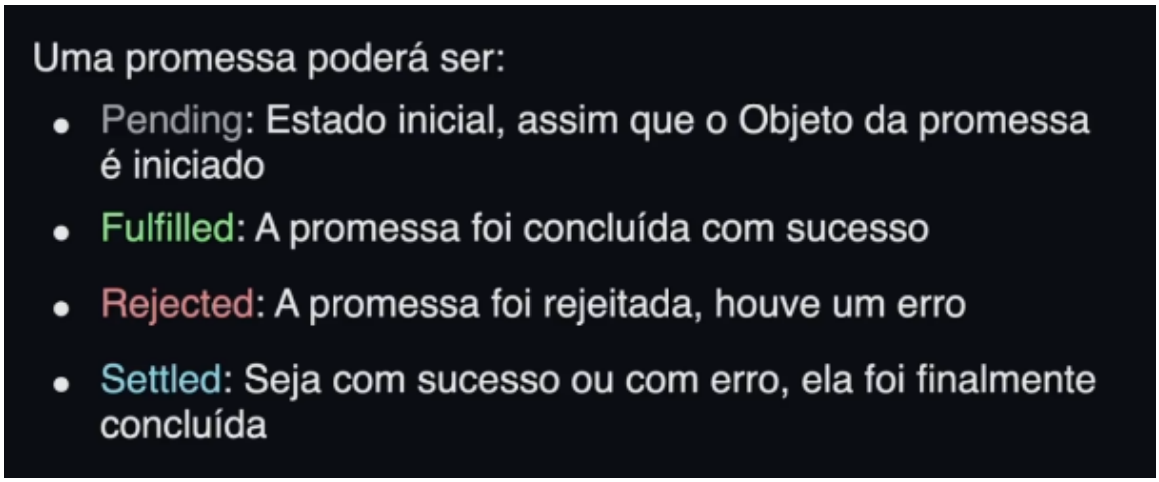
Javascript Assíncrono

Assincronia: partes do código são executadas simultâneamente.

Promise

É um objeto criado para fazer com que alguma ação fique pendente no código. Uma promessa de que algo irá aconter.

Uma promessa pode ter varios status diferentes.



Promise é um objeto, então ela deve ser criada usando

```
new Promise(( resolve, reject ) => {codigo da funcao callback})
```

Geralmente essa função callback avalia alguma condição e retorna um status **resolve()** ou **reject()**, por isso os argumentos **resolve** e **reject.**

Esse objeto promise possui alguns métodos.

`promessa.then(result => console.log( result ) )`: Executa a função callback passada no argumento quando a Promise retorna um resolve. O result é o argumento passado no resolve de dentro da promise quando ele é executado.

Quando o promise da erro, o then não mostra nada.

`promessa.catch(erro => console.log(erro))`: Executa a função callback passada no argumento quando a Promise retorna um erro. O erro é o argumento passado no reject de dentro da promise quando ele é executado.

Quando o promise dá sucesso, o catch não mostra nada.

`promessa.finally( função callback )`: Executa a função passada em argumento quando a promise termina, independendo do status retornado.

Fetch

Fetch é uma função nativa do node js que faz uma requisição dá página pelo https.

```
fetch( URL )
```

Esse fetch é uma promise, é uma função assincrona que pode dar certo ou não.

Axios

Biblioteca de browser e node js para fazer requisições de apis/links no JS.

axios.get(URL) → Devolve uma promise, e essa promise retorna um objeto da página se der sucesso.

Multiplas promises

É possível fazer múltiplas promises como o fetch ao usar o método `Promise.all([])`

`Promise.all([])` retorna um objeto do tipo promise em que sua response é um array que possui a response dos gets/fetchs (promises no geral) feitos dentro dela.

Exemplo

```
import axios from "axios"

const promessas = Promise.all([
  axios.get("https://api.github.com/users/gabriellst")
  axios.get("https://api.github.com/users/gabriellst/repos")
])

promessas.then(respostas => {
  console.log(repostas)
})
```

O argumento respostas do `promessas.then` recebe (por padrão da função then) o argumento passado no resolve da declaração do objeto promise retornada pelo get quando ele for concluido com sucesso.

Async/Await → Syntactic sugar pra promises.

Podemos encapsular o processo da criação de promises por meio de funções async. Essas funções tornam o uso de `.then .catch .finally` menos necessárias, ao passo que, encapsula em uma função assincrona assim como é o setInterval()

A criação dessas funções dão uma ideia de sincronismo, mas, usando a palavra chave `await`, **pode-se esperar as requisições das promises** antes de dar continuidade ao código, tornando o código assíncrono.

Para fazer essas funções assíncronas esperarem o resultado de um promise para continuar

Criando função assíncrona.

```
import axios from "axios"

async function start() {
  const resultado = await axios.get("https://api.github.com/users/gabriellst")
  console.log(resultado)
}
```

Pode-se implementar o uso de try/catch e finally.

```
import axios from "axios"

async function start() {
  try {
    const resultado = await axios.get("https://api.github.com/users/gabriellst")
    console.log(resultado)
  }
  catch(erro) {
    console.log(erro)
  }
  finally {
    console.log("terrminado)
  }
}
```

Como o resultado de cada await é uma promise, podemos usar .then .catch .finally

Spread Operator

O spread operator `…`é um operador que, dado um objeto iterável, como listas ou dicionários, ele espalha seu conteúdo interior em uma linha só, com vírgulas.

Ele desempacota, tira os colchetes `[]` ou chaves `{}` e espalha o conteúdo.

Exemplo

Um array [1, 2, 3, 4] vira 1, 2, 3, 4

Sintaxe: `…objetoIteravel`

Útil para useStates no reactJs

Map Method

Map é um método para objetos iteraveis o qual permite que você aplique uma função em cima de cada elemento do array.

Dentro da função, retorne o valor para fazer uma atribuição ou programe qualquer bloco de código.

Ele pega cada elemento do array, mas também pode pegar o index como o enumerate do python.

```
const array = ["aaaa", "bbbb", "cccc"]
```

```
const array2 = array.map( (element) => {return element + 2})
```

Unary plus

Se colocar um + antes de uma variável, ele é convertido para um numero.

Nullish Coalescing Operator

Similar ao operador ternário

Estrutura: `(variavel1 ?? variavel2)`

Caso a variave1 seja nula ou undefined, ele preenche com a variavel 2, caso contrário, preenche com a váriavel 1.

Desestruturação de objetos.

Dado um objeto, é possível extrair variáveis desse objeto usando a desestruturação.

Se eu tenho um objeto tal que

```
const objeto = {
  name: "Gabriel",
  idade: "20",
  curso: "cdia"
}
```

É possível extrair esses atributos para variaveis específicas, sem ter que usar o `objeto.atributo` ao referenciá-las.

```
const objeto = {
  name: "Gabriel",
  idade: "20",
  curso: "cdia"
}
const { nome, curso } = objeto
```

A variável nome vai extrair **nome** do objeto, assim como curso.

Optional Chaining ?.

Optional chaining é um operador que indica que um atributo/método pode ser opcional, existir ou não.

Isso evita erros na chamada de atributo.

Se não existir, retorna um undefined, então pode ser usado com o Nullish Coalescing Operator para fazer operações similares ao operador ternário.