JS PRO

Próximos passos:

- ES6
- Let statement [1-3]
- Const statement [4]
- Template Literals [5-9]
- Arrow function [10-15]
- Spread Operator [16-21]
- DOM [22-24]
- Eventos [25]
- Projeto 1 [26]
- Requisições HTTP [27]
- Projeto 2 [28]

LET

let permite criar variáveis com escopo de bloco

```
let myVar = 'Some value'
let anyNumber = Math.random()
let iAmTrue = true
let emptyObj = {}
let emptyArray = []
```

LET

Então qual a diferença pro var?

```
var greeting = 'Olá!'
console.log(greeting)

greeting = 'Olá aqui de baixo'
console.log(greeting)

var greeting = 'Posso usar var de novo'
console.log(greeting)
```

LET

Lembra do escopo em bloco?

```
var firstName = 'Leonardo'

if (true) {
  var lastName = 'Redin'
}

var fullName = firstName + ' ' + lastName

console.log(fullName)
```

CONST

Declarar uma **const** cria um valor que é uma referência de **somente leitura.**
Ela pode ser tanto global ou local a função ou bloco em que foi definida.
Dessa forma ela deve ser **obrigatoriamente** inicializada.

CONST

```
const firstName = 'Leonardo'

//erro firstName = 'Redin'

//péssima prática já que não conseguimos alterar o valor
const lastName

console.log(lastName)
```

Template Literals

Template literals são usadas com as aspas(` `) ao invés de ' '
ou " "
Elas podem conter expressões JS dentro.
Todas as partes, incluindo o resultado dessa(s) expressão(ões)
são concatenadas e viram uma única string

Template Literals

```
const myName = 'Leonardo'
const myLastName = 'Redin'

const fullName = `Meu nome completo é: ${myName} ${myLastName}`

console.log(fullName)
```

Template Literals

```
const arrayOfNames = ['Leonardo', 'Redin']

const fullName = `Nome completo: ${arrayOfNames.join(' ')}`

console.log(fullName)
```

Template Literals

```
const myName = `Meu nome é: ${(function () {
  const first = 'Leonardo'
  const last = 'Redin'

  return `${first} ${last}`
})()}`

console.log(myName)
```

Arrow functions

Arrow function é uma maneira diferente de se declarar uma Function Expression.
Isso implica em uma grande mudança: o escopo do 'this'.
Representada pelo sinal de '=>'

Arrow functions

```
// ES5
var es5Func = function () {
  return 'whatever'
}

// ES6

const es6Func = () => 'whatever'
```

Arrow functions

ES6

- menos código
- sem a necessidade da palavra function
- () são opcionais às vezes
- retorno implícito sem as chaves {}

Arrow functions

Diferentes maneiras declarar arrow functions

Arrow functions

1 parâmetro

```
let quadrado

quadrado = (num) => num * num
quadrado = (num) => {
  return num * num
}
```

Arrow functions

2 ou mais parâmetros

```
let multiply

multiply = (x, y) => x * y
```

Spread operator

A sintaxe de propagação (Spread) permite que um objeto iterável, como um array ou string, seja expandida em locais onde zero ou mais argumentos (para chamadas de função) ou elementos (para literais de array) sejam esperados ou uma expressão de objeto seja expandida em locais onde zero ou mais pares de chave-valor (para literais de objeto) são esperados. (MDN)

Spread Operator - Arrays

```
const pokemons = ['Bulbasaur', 'Charmander', 'Squirtle']

console.log(pokemons)

console.log(...pokemons)
```

Spread Operator - Arrays

```
const firstGen = ['Bulbasaur', 'Charmander', 'Squirtle']

const secondGen = ['Chikorita', 'Cyndaquil', 'Totodile']

const joined = firstGen.concat(secondGen) // retorna um novo array
```

Spread Operator - Arrays

ES6

```
const firstGen = ['Bulbasaur', 'Charmander', 'Squirtle']

const secondGen = ['Chikorita', 'Cyndaquil', 'Totodile']

const joined = [...firstGen, ...secondGen]

console.log(joined)
```

Spread Operator - Functions

```
const firstGen = [1, 4, 7]

const secondGen = [2, 5, 8]

function add(num1, num2, num3) {
  const result = num1 * num2 * num3
  console.log(result)
}

add(firstGen[0], firstGen[1], firstGen[2])
add(secondGen[0], secondGen[1], secondGen[2])
```

Spread Operator - Functions

ES6

```
const firstGen = [1, 4, 7]

const secondGen = [2, 5, 8]

function add(num1, num2, num3) {
  const result = num1 * num2 * num3
  console.log(result)
}

add(...firstGen)
add(...secondGen)
```

DOM (Document Object Model)

O Modelo de Objeto de Documento (DOM) é uma interface de programação para documentos HTML, XML e SVG . Ele fornece uma representação estruturada do documento como uma árvore. O DOM define métodos que permitem acesso à árvore, para que eles possam alterar a estrutura, estilo e conteúdo do documento. O DOM fornece uma representação do documento como um grupo estruturado de nós e objetos, possuindo várias propriedades e métodos. Os nós também podem ter manipuladores de eventos que lhe são inerentes, e uma vez que um evento é acionado, os manipuladores de eventos são executados. Essencialmente, ele conecta páginas web a scripts ou linguagens de programação. (MDN)

DOM - Window Object

```
console.log(window.innerWidth)
console.log(outerWidth)
console.log(innerHeight)
console.log(outerHeight)
console.log(window)
localStorage e sessionStorage
document
location
```

DOM - Document

- title
- children
- como modificar o próprio DOM
- adicionando estilos
- pegando os elementos do DOM
- criando elementos com JS
- inserindo novos elementos no DOM
- deletar elementos

DOM

```
const title = document.title
const children = document.children
document.title = 'Escolha um título'
const h1 = `
  <h1>
    Olá Target
  </h1>
`

const app = document.getElementById('app')

app.innerHTML = h1
```

Eventos

Vamos para o codesandbox :)

- onclick
- eventListener
- target

Projeto 1 - TODO App

Funcinalidades:

- Criar um TODO
- Deletar um TODO
- Limpar a lista
- Buscar TODO
- Remover espaços em branco das tarefas
- Limpar formulário após submissão
- Não permitir tarefa em branco
- Persistir dados usando localStorage (Bônus)

HTTP Requests

Vamos para o codesandbox :)

- Tipos de requisições
- GET / POST
- Fetch API

Projeto 2 - App de Clima