# O que é JavaScript

Programação de alto nível.

Serve para criar scripts dinâmicos que realizam interações de apps ou páginas web.

# Entendendo variáveis e seus valores

## Tipagem

Funciona como uma regra de uso de dados, quanto mais forte a tipagem mais obrigatório é a declaração do tipo de dado.

Tipagem em JS é fraca e a declaração acontece de forma dinâmica.

Ex.: Ao criar uma variável, o JS já converte em string.

## Tipos primitivos

|  |  |
| --- | --- |
| **String** | Cadeia de caracteres (servirá como texto); |
| **Number** | Números, ficam azul no console; |
| **Null** | Nada/Vazio; |
| **Boolean** | Verdadeiro ou Falso; |
| **Undefined** | O computador não conseguiu encontrar um valor. |

Var – escopo global e local, pode ter o valor alteado. Se não tiver um valor inicial será tratado como null.

Let – escopo local de bloco, pode ter o valor alterado. Se não tiver um valor inicial será tratado como null.

Const – escopo local de bloco, comente leitura, o valor inicial é obrigatório e não pode ser alterado.

**Escopo global**: Quando a variável é declarada fora de qualquer bloco, sua visibilidade fica disponível em todo o código.

**Escopo local**: quando a variável é declarada dentro de um bloco, sua visibilidade pode ficar disponível ou não.

|  |  |
| --- | --- |
| **==** | Igual a; (compara valores, retornando um boolean)  Ex.: 3 número e “3” letra são iguais com o ==. |
| **===** | Valor igual e igual. (compara o valor e o tipo);  Ex.: 3 número e “3” letra. São diferentes com o ===. |

## Operadores relacionais

|  |  |  |  |
| --- | --- | --- | --- |
| > | < | >= | <= |

## Operadores lógicos

|  |  |  |
| --- | --- | --- |
| && - “e” | || - “ou” | ! – “não” |
|  |  |  |

# Vetores e Objetos

## Vetores ou Arrays

São um tipo de lista ou matriz de variáveis, onde cada variável possui um índice.

## Manipulando Arrays

|  |  |
| --- | --- |
| forEach () | Executa uma dada função em cada elemento de um array.  Ex.: array.forEach (function (item, index) {console.log (item, index) } ) |
| Push () | Adiciona itens no final do array; |
| Pop () | Remove item no final do array; |
| Shift () | Remove item no início do array; |
| Unshift () | Adiciona item no início do array; |
| indexOf () | Retorna o índice de um valor; |
| Splice () | Remove ou substitui um item pelo índice; |
| Slice () | Retorna uma parte de um array existente. |

## Objetos

Dados que possuem propriedades e valores que definem suas características. Devem ser declarados “{ }”.

## Manipulando objetos

Propriedade de obetos podem ser atribuídas à variáveis, facilitando a manipulação do objeto. Chamamos isso de desestruturação.

Var xicara = { cor: ‘azul’, tamanho: ‘p’, função: tomarCafe() }

Var cor = xicara.cor

Var tamanho = xicara.tamanho

Var função = tomarCafe()

Outra forma de fazer a desestruturação é utilizando chaves ao declarar a variável.

Var xicara = { cor: ‘azul’, tamanho: ‘p’, função: tomarCafe() }

Var {cor, tamanho, função} = xicara

# Estruturas condicionais

São instruções para realizar determinadas tarefas a partir de uma condição.

## If ternário

Quando faz uma verificação em uma única linha.

A<18 ? console.log (‘Menor de idade.” : console.log (‘Maior de idade.’)

## Switch/case

Funciona como uma estrutura condicional também.

|  |  |
| --- | --- |
|  | Ex.:  Switch (${expressão} {  Case placar>1:  Console.log (‘ texto1’)  Break  Case placar<1:  Console.log (‘ texto2’)  Break  } |

## Laços de repetição

Estruturas condicionais que repetem uma instrução até atingir determinada condição.

|  |  |
| --- | --- |
| For | Funciona como uma repetição de instrução até que a condição seja falsa. |
| For/in | Funciona como uma repetição a partir de uma propriedade.  Ex.: For (i in lista) { } |
| For/of | Funciona como uma repetição a partir de um valor.  Ex.: for (i of lista) { } |
| While | Executa uma instrução ‘enquanto’ determinada condição for verdadeira. |
| Do/While | Executa uma instrução “até que” determinada condição seja falsa, a verificação é feita depois da execução.  Do {  A++  Console.log(a)  } while (a<10) |

# Funções e suas particularidades

## Funções

São blocos de comandos e instruções para a execução de determinada tarefa.

## Funções com parâmetros

As funções podem receber em sua declaração, parâmetros, que servem como variáveis, onde sua atribuição pode ser feita durante a chamada da função.

## Expressões de funções

São funções atribuídas à expressões. A nomeação das funções por expressão é opcional.

Var função = function ( ) {

Console.log (“Sou uma mensagem de expressão”) }

Função ( )

## Arrow function

São funções de expressão de sintaxe curta. Sempre serão anônimas e portanto não podem ser nomeadas. Deve ser declarada com parênteses “( )”, seguido de “=>” e depois chaves “{ }”.

Var função = ( ) => {

Console.log(“Sou uma arrow function”) }

Função ()

10 – Funções

Uma variável declarada dentro de uma função só poderá ser utilizada dentro dela.

Ao invocar “return” a função para de ser executada.

## Função anônima

Funções que representam expressões. Uma variável pode armazenar uma função

      const soma = function (*a*,*b*){

        return a + b

      }

## Função auto invocável

Uma função anônima entre parênteses, seguida por outro par de parênteses, que representa sua chamada.

(     function () {

        let name = 'Digital Innovation One'

        return name

      }

)()

## Call-backs

Uma função passada como argumento para outra.

      const calc = function (*operação*, *num1*, *num2*){

        return operação (num1, num2)

        }

        const soma = function (*num1*, *num2*){

          return num1 + num2

        }

        const sub = function(*num1*, *num2*){

          return num1 - num2

        }

        const resulSoma = calc(soma, 1, 2)

        const resulSub - calc(sub, 1, 2)

        console.log(resulSub)

        console.log(resulSoma)

## Valores padrão

      function exponencial (*array*, *num* = 1){

        const result = []

      }

Obs.: irá aplica o valor 1 em “num”, caso não for adicionado algum número para o array.

## Objeto ‘Arguments ‘

Um array com todos os parâmetros passados quando a função foi invocada.

      function findMax(){

        let max = -Infinity

        for (let i=0; l<*arguments*.length; i++){

          if (*arguments*[i]>max){

            max = *arguments*[i]

          }

        }

        return max

      }

# Array e Objetos em funções

# Arrays

Spread: Uma forma de lidar separadamente com elementos.

      function sum(*x*,*y*,*z*){

        return x+y+z

      }

      const numbers = [1,2,3]

      console.log(sum(...numbers));

o que era parte de um array se torna um elemento independente.

Rest: combina os argumentos em um array.

*Function* confereTamanho(...args) {

        Console.log (args.length)

      }

      confereTamanho ( )

      confereTamanho (1, 2)

      confereTamanho (3, 4, 5)

O que era um elemento independente se torna parte de um array.

# Objetos

Object Destructuring: Entre chaves { } podemos filtrar apenas os dados que nos interessam em um objeto. É uma expressão JavaScript que possibilita extrair dados de arrays ou objetos em variáveis distintas.

var array = [{ nome: 'Jaime', idade: 30}, nome: 'Raynara', idade: 25]

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

    const {nome, idade} = array[i]

    console.log(nome)

    console.log(idade)

vai imprimir os nomes dentro do array e depois as idade, não havendo a necessidade de escrever “array[ i ].nome” ou “array[ i ].idade”.

      Const user = {

        id: 42,

        displayName: 'jdoe',

        fullName: {

          firstName: 'John',

        lastname: 'Done'

      }}

      function userId (*id*) {

        return id

      }

      function getFullName ({fullName:{firstName: *first*, lastName: *last*}}){

        return `${first} ${last}`

      }

# IF / Else

Maneiras diferentes de escrever uma mesma função.



## Aninhar

function numeroPositivo(*num*){

  const ehNegativo = num < 0

  const maiorQueDez = num > 10

  if (ehNegativo){

    return "Esse número é negativo!"

  } else if (!ehNegativo && maiorQueDez){

    return "Esse número é positivo e maior que 10!"

  }

  return "esse número é positivo!"

  }

# For

## For...In

Loop entre propriedades enumeráveis de um objeto.

*Function* exemplo(obj){

        for (prop in obj){

          console.log(prop)

        }

      }

      const meuObjeto = {

        nome: "João",

        idade: "20",

        cidade: "Salvador"

      }

      exemplo (meuObjeto)

para imprimir a propriedade do objeto:

nome

idade

cidade

*Function* exemplo(obj){

        for (prop in obj){

          console.log(obj[prop])

        }

      }

      const meuObjeto = {

        nome: "João",

        idade: "20",

        cidade: "Salvador"

      }

      exemplo (meuObjeto)

para imprimir o valor:

joão

20

Salvador

## For...of

Loop entre estruturas iteráveis (arrays, strings)

      var array = [1, 3, 4, 6, 80, 33, 23, 90]

*Function* logLetras (palavra) {

        for (letra of palavra) {

          console.log (letra)

        }

      }

      const palavra = "abacaxi"

      logLetras (palavra)

vai imprimir cada letra separada de “abacaxi”

      var array = [1, 3, 4, 6, 80, 33, 23, 90]

*Function* logNumeros (palavra) {

        for (letra of palavra) {

          console.log (letra)

        }

      }

      const nums = [30, 20, 233, 2]

      logNumeros (nums)

vai impimir:

30

20

233

2

# While e Do While

## While

Executa instruções até que a condição se torne falsa.

while (num <=5){

    console.log(num)

num++

}

## Do While

Executa instruções até que a condição se torne falsa. Porém, a primeira execução sempre ocorre.

do {

    console.log(num)

    num++

while (num <=5){

}}

## This

É uma referência de contexto. No exemplo, this refere-se ao objeto pessoa.

const pessoa = {

    firstName: "André",

    lastName: "Soares",

    id: 1,

    fullName: function(){return this.firstName + " " + this.lastName},

    getId: function(){

        return this.id

    }

}

O ‘this’ no exemplo acima se refere ao objeto “pessoa”. É como se tivesse escrito “pessoa.firstName” no lugar de “this.firstName”.

|  |  |
| --- | --- |
| O valor do “this” pode mudar de acordo com o lugar no código onde foi chamada. | |
| **Contexto** | **Referência** |
| Em um objeto (método) | Próprio objeto dono do método |
| Sozinha | Objeto global (em navegadores, window) |
| Função | Objeto global |
| Evento | Elemento que recebeu o evento |

# Manipulando seu valor (call e apply)

## Call

const pessoa = {

    nome 'Miguel'

}

const animal = {

    nome: 'Murphy'

}

function getSomething(){

    console.log(this.nome)

}

getSomething.call(pessoa)

irá imprimir: Miguel

O “this” irá chamar “pessoa” graças ao “getSomething.call (pessoa)”.

É possível passar parâmetros para essa função separando-os por vírgulas.

const myObj = {

    num1: 2,

    num2: 4,

}

function soma(a,b){

    console.log (this.num1 + this.num2 + a + b)

}

soma.call (myObj, 1, 5)

Vai imprimir 12.

## Apply

É possível passar parâmetros para essa função dentro de um array.

const pessoa = {

    num1: 2,

    num2: 4,

}

function soma(a, b) = {

console.log(this.num1 + this.num2 + a + b)

}

soma.apply (pessoa, [1, 5])

vai imprimir: 12

## Bind

Clona a estrutura da função é chamada e aplica o valor do objeto passado como parâmetro.

Const retornaNomes = function () {

    Return this.nome

}

let bruno = retornaNomes.bind ({nome: 'Bruno'})

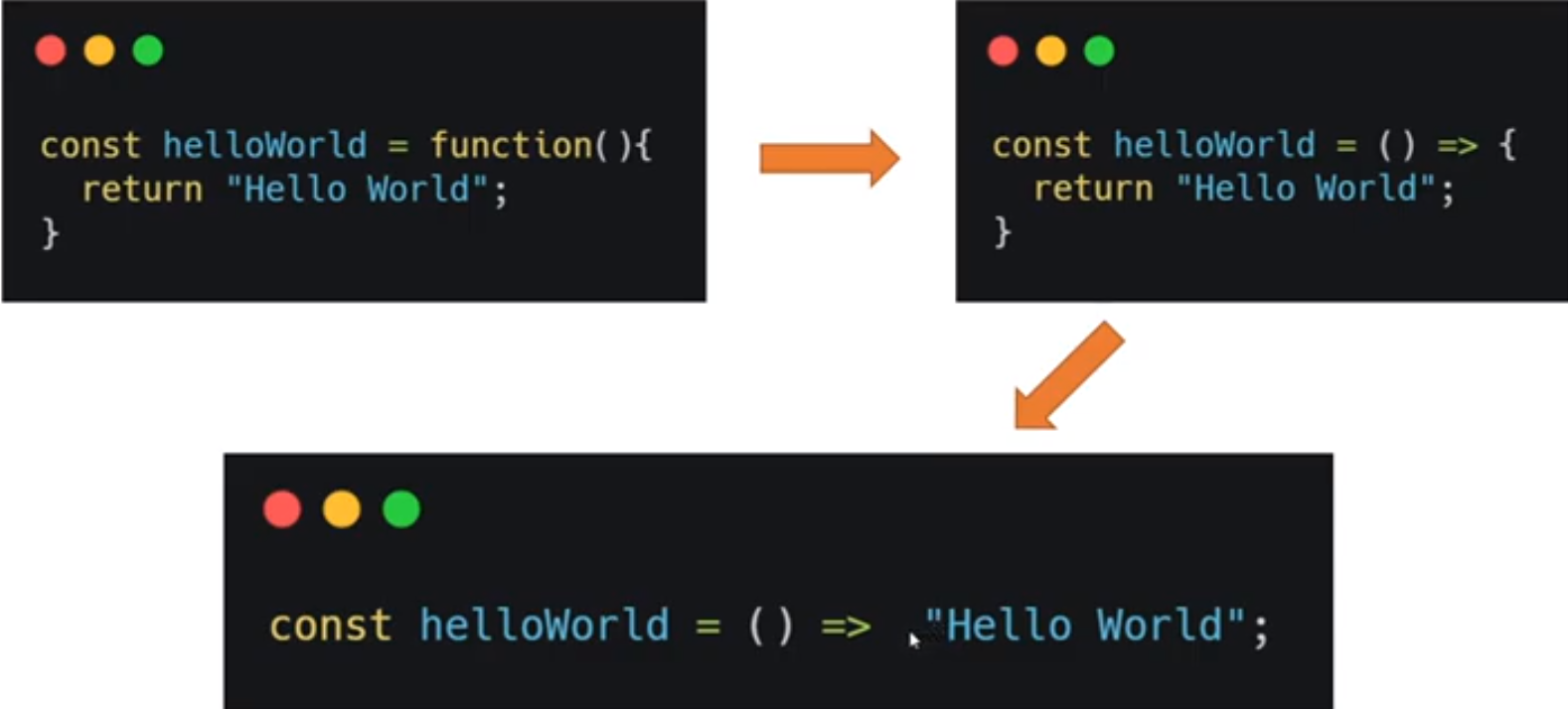
bruno ()

vai imprimir: Bruno

# Arrow functions =>

Sintaxe

É o mesmo que:



Caso exista apenas uma linha, pode dispensar as chaves e o return;

Caso exista apenas um parâmetro, pode dispensar os parênteses.

Arrow function **NÃO** faz HOISTING

**Algumas restrições**

* “this” sempre será o objeto global. Métodos para modificar seu valor não irão funcionar;
* Não existe o objeto “arguments”;
* O construtor (ex: new **MeuObjeto ()** ) também não pode ser utilizado.

Coleções

# Map

const myMap = new Map ()

Estrutura do MAP:

* Uma coleção de arrays no formato [chave, valor]
* Pode iterado por um loop for..of

Métodos do MAP:

|  |  |
| --- | --- |
| Adicionando “apple” como chave e o valor “fruit” | Const myMap = new Map( )  myMap.set (‘apple’, ‘fruit’) |
| Checar o valor de “apple” | myMap.get (‘apple’) |
| Deletar um valor | myMap.delete(‘apple’) |

Diferença de MAP vs Objeto

* Maps podem ter chaves de qualquer tipo, já o objeto, não.
* Maps possuem a propriedade length
* Maps são mais fáceis de iterar
* Utilizado quando o valor das chaves é desconhecido
* Os valores tem o mesmo tipo

# Set

Estrutura de um SET

São estruturas que armazenam apenas valores únicos.

const myArray = [1, 1, 2, 2, 3, 4, 5, 6, 7, 8, 2, 1]

const mySet = new Set (myArray)

|  |  |
| --- | --- |
|  |  |
| Adicionar um valor | mySet.add (5) |
| Encontrar um valor dentro | mySet.has (1) |
| Deletar um valor | mySet.delete (5) |

Diferença de SET vs ARRAY

* Possui valores únicos;
* Em vez da propriedade length, consulta-se o número de registros pela propriedade size;
* Não possui os métodos map, filter, reduce etc.

Debugging e Error Handling

# Tipos de erros

## ECMAScript Error

Composto por:

* Mensagem
* Nome
* Linha
* Call stack

## DOMException

Erros relacionados ao Document Object Model (DOM)

# Tratamento de erros

## Throw

Use a instrução throw para lançar um erro. Quando você lança uma exceção, expressão especifica o valor da exceção, que no caso, será um erro.

## Try...Catch

As declarações **try...catch** marcam um bloco de declarações para testar (**try**), e especifica uma resposta (digitada por vc), caso uma exceção seja lançada.

Dentro do bloco try são executadas instruções e caso ocorra algum erro durante o processo, ele será tratado no bloco catch.

        function verificaPalindromo(string){

            if (!string) throw "String inválida"

            return string === string.split('').reverse().join('')

        }

        function tryCatchExemplo (string){

            try{

                verificaPalindromo(string)

            }

            catch(e){

                console.log(e)

            }

        }

        tryCatchExemplo('')

vai imprimir: String inválida

## Finally

O método **finally()** retorna uma [Promise](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Promise). Quando a promise for estabelecida, tenha ela sido realizada ou rejeitada, executa-se a função callback especificada. Isso permite a execução de um código que acontecerá independentemente da Promise ter sido realizada (com sucesso) ou rejeitada (com falha).

        function verificaPalindromo(string){

            if (!string) throw "String inválida"

            return string === string.split('').reverse().join('')

        }

        function tryCatchExemplo (string){

            try{

                verificaPalindromo(string)

            }

            catch(e){

                console.log(e)

            }

            finally{

                console.log('A string enviada foi: ' + string)

            }

        }

        tryCatchExemplo('')

vai imprimir:

String inválida

A string enviada foi:

# Criando erros

Como manipular um objeto error.

new Error(message, fileName, lineNumber)

const MeuErro = new Error ('Mensagem Inválida')

throw MeuErro

vai imprimir o erro

Alterando a mensagem do erro

const MeuErro = new Error ('Mensagem Inválida')

MeuErro.name = 'Ocorreu um erro aqui rapaz'  //Altera a mensagem do erro

throw MeuErro

|  |  |
| --- | --- |
| MeuErro.name | Mostra o nome do erro |
| MeuErro.stack | Vai mostrar onde ocorreu o erro |
| MeuErro | Apresenta tudo |
|  |  |

JavaScript Assíncrono

# Assincronicidade

Assíncrono: Que não ocorre ou não se afetiva ao mesmo tempo.

Síncrona: Ocorre em tempo real. (JavaScript funciona assim)

Promises (promessa)

É um objeto de processamento assíncrono. Inicialmente seu valor é desconhecido. Pode ser resolvida ou rejeitada.

Pode ter 3 estados:

* Pending (pendente);
* Fulfilled (completado);
* Rejected (rejeitado)

Estrutura:

        const myPromise = new Promise ((resolve, reject) =>{window.setTimeout(() => {

            resolve(console.log('Resolvida!'))

        }, 2000)

        })

## Async e Await

Async: assíncrona.

Funções assíncronas precisam dessas duas palavras chave.

# Consumindo APIs

API é uma forma de intermediar os resultados do back-end com o que é apresentado no front-end.

Você consegue acessá-la por meio de URLs.

Application Programming Interfaces

JSON: JavaScript Object Notation

É muito comum que APIs retornem seus dados no formato .json, portanto precisamos tratar esses dados quando os recebermos.

## Fetch

    fetch (copiar a url aqui, options)

    .then(response => response.json())

    .then(json => console.log(json))

    // vai retornar uma Promisse

Retorna uma ‘promise’

Pode fazer operações no banco (post, get, put, delete, etc)

Orientação a Objetos

# Conceito

Paradigmas

Um paradigma pode ser entendido como um **tipo de estruturação ao qual a linguagem deverá respeitar**.

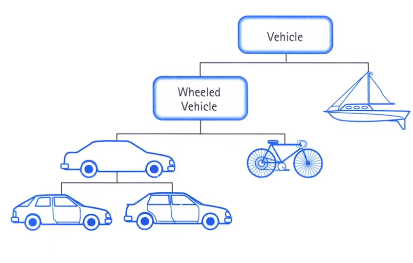
São objetos que possuem uma série de propriedades.

Pilares

* Herança;
* Polimorfismo;
* Encapsulamento;
* Abstração.

Pilares

Abstração: processo mental que consiste em isolar um aspecto determinado de um estado de coisas relativamente complexo, a fim de **simplificar** a sua avaliação, classificação ou para permitir a comunicação do mesmo.



Herança: O objeto filho herda propriedades e métodos do objeto pai.

Encapsulamento: cada classe tem propriedades e métodos independentes do restante do código.

Polimorfismo: Objetos podem herdar a mesma classe pai, mas se comportarem de forma diferente quando invocamos seus métodos.

# No JavaScript

## Protótipos

Todos os objetos JavaScript herdam propriedades e métodos de um prototype. O objeto Object.prototype está no topo desta cadeia.

## Classes

Syntatic Sugar: Uma sintaxe feita para facilitar a escrita.

JavaScript não possui classes nativamente. Todas as classes são objetos e a herança se dá por protótipos.

Map, Filter e Reduce

# Map

O método **map()** invoca a função callback passada por argumento para cada elemento do Array e devolve um novo Array como resultado.

* Cria um novo array;
* Não modifica o array original;
* Realiza as operações em ordem.

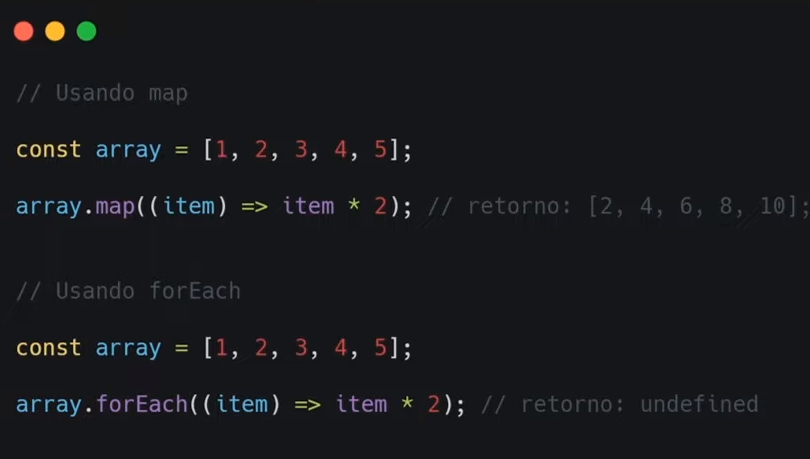
array.map(callback, thisArg)

**call-back**: função a ser executada em cada elemento.

**thisArg (opcional):** valor de ‘this’ dentro da função de call-back.

## Diferença entre MAP com forEach

* Valor de retorno diferente;
* Considera se um array auxiliar será necessário.



# Filter

O método **filter()** cria um novo array com todos os elementos que passaram no teste implementado pela função fornecida.

* Cria um novo array;
* Não modifica o array original.

    array.filter (callback, thisArg)

exemplo:

        cont frutas = ['maçã fuji', 'maã verde', 'laranja', 'abacaxi']

        frutas.filter((fruta) => fruta.includes('maça'))

        // vai retornar: ['maçã fuji', 'maça verde']

# Reduce

O método **reduce()** executa uma função **reducer** (fornecida por você) para cada elemento do array, resultando num único valor de retorno.

Executa uma função em todos os elementos do array, retornando um valor único.

        const callbackFn = function (acumulator, currentValue, index, array)

        array.reduce (callbackFn, initialValue)

**call-back**: função a ser executada a partir do acumulador.

**thisArg (opcional):** valor sobre o qual o retorno final irá atuar.

**Acumulator/prevValue**: acumulador de todas as chamadas de callbackFn.

**currentValue**: elemento atual sendo acessado pela função.

Manipulando a D.O.M com JavaScript

# O que é o DOM

DOM (Document Object Model): É um padrão de como acessar e modificar os elementos HTML de página.

DOM vs BOM

BOM (Browser Object Model): Tudo que está dentro do objeto window. (árvore de dependências do Browser.

# Selecionando elementos

Document.querySelectorAll (‘.primeira-classe .segunda-classe’) - serve para filtrar um elemento

|  |  |
| --- | --- |
| Método | Descrição |
| document.createElement (element) | Cria um novo elemento HTML |
| document.removeChild (element) | Remove um elemento |
| document.appendChild (element) | Adiciona um elemento |
| document.replaceChild (new, old) | Subtitui um elemento |

# Trabalhando com estilo

Para adicionar um estilo para uma classe é usado o “elemento.classList”.



|  |  |
| --- | --- |
| classList.add | Adiciona a classe “novo-estilo” |
| classList.remove | Remove a classe “classe” |
| classList.toggle | Adiciona a classe “dark-mode” caso ele não exista, se existir, então vai remover. |

Usando diretamente o CSS de um elemento:

Document.getElementsByTagName (“p”).style.color = “blue”

# Eventos

|  |  |
| --- | --- |
| Eventos de mouse | |
| Mouseover | Quando o mouse ficar em cima do objeto |
| mouseout | Quando o mouse sair do objeto |
| Click | Clicar. |
| Dbclick | Duplo clique. |
|  |  |
|  | Eventos de atualização |
| change | Quando houver mudança |
| load | Quando a página carregar |

## Event Listener

Serve para acionar o evento diretamente pelo JavasCript quando realizar uma determinada ação.

        const botão = document.getElementById ("meuBotao")

        botão.addEventListener ("click", outraFunção)

## atributo HTML

especifica a função a ser chamada diretamente no elemento HTML.

    <h1 onclick="mudaTexto(this)">Clica aqui!</h1>

    <script>

        function mudaTexto(id){

            id.innerHTML = "Mude!"

        }

    </script>