

- **Operadores Bitwise em Javascript**

Um bitwise operador realiza operações diretamente nos valores binários que estão sendo operados. As operações comuns são as operações básicas da lógica booleana.

Todas as operações bitwise vão passar pelos bits um a um do primeiro valor comparado e vão comparar com a mesma posição do segundo valor.

Independente se tenho equivalência ou não, sempre vai retornar o bit 1

EX:

```
Let n1 = 14, Let n2 = 11
```

```
Let res = n1 & n2
```

```
Console.log(res)
```

```
Resultado: 10
```

11 (DECIMAL - 1011)

14 (DECIMAL - 1110)

OPERAÇÃO -> 1010

- **Operador ternário**

EX1:

```
Let num = 11
```

```
Res = (!(num%2) ? "Par" : "Impar")
```

```
Console.log(res)
```

Resultado -> IMPAR

```
//0 = FALSE
```

```
//1 = TRUE
```

```
//! = NEGAÇÃO
```

```
//Teste lógico ? se verdadeiro : se falso
```

EX2:

```
Let num1 = 10
```

```
Let num2 = 5
```

```
Res = (num1 > num2 ? "Verdadeiro" : "Falso")
```

```
Console.log(res)
```

Resultado -> Verdadeiro

- **Operador Typeof**

O operador Typeof pega a variável que estamos colocando como parâmetro e me retorna o tipo de dado dessa variável.

EX:

```
Let v1 = 10
```

```
Let v2 = "10"
```

```
Let v3 = v1 === v2
```

```
Let v4 = {nome: "Wellington"}
```

```
Console.log(typeof(v1) //number
```

```
Console.log(typeof(v2) //string
```

```
Console.log(typeof(v3) //boolean
```

```
Console.log(typeof(v4) //object
```

- **Operador Spread**

O operador spread permite que você espalhe elementos de um objeto iterável, como um array, um map, ou um conjunto.

Sintaxe -> ---

EX1:

```
const jogador1 = {nome: "Bruno", energia: 100, vidas: 3, magia: 150}
```

```
const jogador2 = {B: "Bruce", energia: 100, vidas: 5, velocidade: 80}
```

```
const jogador3 = {...jogador1, ...jogador2 }
```

```
console.log(jogador3)
```

Resultado -> {nome: 'Bruce', energia: 100, vidas: 5, magia: 150, velocidade: 80}

EX2:

```
Const soma = (v1,v2,v3) => {  
    Return v1 + v2 + v3  
}  
Let valores = [1,5,4]  
Console.log(soma(...valores))  
Resultado -> 10
```

EX3:

HTML

```
<div>Wellington</div>  
<div>CfbCursos</div>
```

```
Const objs1 = document.getElementsByTagName("div")  
Const objs2 = [...document.getElementsByTagName("div")]
```

```
Console.log(objs1) //HTMLCollectionOf  
Console.log(objs2) //Array
```

```
objs2.forEach(elemento => {  
    Console.log(element)  
}); //Só é possível percorrer o 'objs2' porque ele agora é um array  
grças ao spreand.
```

- **Loops FOR IN e FOR OF**

FOR IN

O laço **for...in** interage sobre propriedades enumeradas de um objeto, na ordem original de inserção. O laço pode ser executado para cada propriedade distinta do objeto.

EX:

```
Let num = [10,20,30,40,50]
```

-For comum-

```
For(let i = 0; i < num.length; i++){  
    Console.log(num[i])  
}
```

Resultado: 10,20,30,40,50

-For in-

```
For(n in num){  
    Console.log(num[n])  
}
```

Resultado: 10,20,30,40,50

FOR OF

A instrução JavaScript **for of** percorre os valores de um objeto iterável.

Ele permite que você faça um loop sobre estruturas de dados iteráveis, como Arrays, Strings, Maps, NodeLists e muito mais:

(O FOR OF INTERAJE DIRETAMENTE COM OS OBJETOS DA COLEÇÃO, DIFERENTE DO FOR IN)

EX:

```
Let num = [10,20,30,40,50]
```

```
For(n of num){  
    Console.log(n)  
}
```

Resultado: 10,20,30,40,50

- **BREAK e CONTINUE**

Break

Break é uma expressão que vai gerar uma interrupção na execução.

(Encerra o loop e dar continuidade ao programa)

Continue

Cancela a interação atual do loop e pula pra próxima interação.

(Continua a execução do loop)

- **Parâmetros REST em funções**

Um **Spread** o transforma em um parâmetro REST, assim podendo inserir a quantidade de parâmetro automático.

EX:

```
Function soma(...valores){ //Valores virou um array
    Return valores.length
}
```

```
Console.log(soma(10,5,2))
```

Resultado: 3 //Aqui soma a quantidade de parâmetros por conta do length.

Usando o for para operar esses valores

EX2:

```
Function soma(...valores){
    Let tam = valores.length
    Let res = 0
    For(let i = 0; i < tam; i++){
        Res += valores[i]
    }
    Return res
}
```

```
Console.log(soma(10,5,2))
```

Resultado: 17

EX3:

Com FOR OF

```
Function soma(...valores){
    Let res = 0
```

```
    For(let v of valores){  
        Res += v  
    }  
    Return res  
}  
Console.log(soma(10,5,2,8))
```

- **Funções construtora anônima**

EX:

```
Const f = new Function("v1","v2", "return vi + v2") //Função  
Construtor Anônima
```

```
Console.log(f(10,5))
```

Resultado: 15

Obs: O último parâmetro da função sempre será o corpo, ou seja, o que a função deverá fazer.

- **Funções Geradoras**

A declaração `function*` (palavra chave function seguida de um asterisco) define uma função geradora (generator function), que retorna um objeto Generator.

EX:


```
function* generator(i) {  
  yield i;  
  yield i + 10;  
}
```

```
const gen = generator(10);
```

```
console.log(gen.next().value);
```

```
// expected output: 10
```

```
console.log(gen.next().value);
```

```
// expected output: 20
```

EX2:

```
Function* perguntas(){  
  Const nome = yield 'Qual seu nome?'  
  Const esporte = yield 'Qual seu esporte favorito?'  
  Return 'Seu nome é ' + nome + ', seu esporte favorito é ' + esporte  
}
```

```
Const itp = perguntas()
```

```
Console.log(itp.next().value)
```

```
Console.log(itp.next().value)
```

```
Console.log(itp.next().value)
```

- **MAP()**

O método `map()` é invocado a partir de um array e recebe como parâmetro uma função de callback, que é invocada para cada item e retorna o valor do item equivalente no array resultante.

EX:

```
const cursos = ['HTML', 'CSS', 'JAVASCRIPT', 'PHP', 'REACT']  
console.log(cursos)  
  
cursos.map((elemento, indice)=>{  
    console.log('Curso: ' + elemento + ' - Posição do curso: ' + indice)  
})  
//Elemento: (curso - html)  
//Indice: (posição - 0)
```

Resultado:

Curso: HTML - Posição do curso: 0

Curso: CSS - Posição do curso: 1

Curso: JAVASCRIPT - Posição do curso: 2

Curso: PHP - Posição do curso: 3

Curso: REACT - Posição do curso: 4

EX2:

HTML

```
<div id="c1">HTML5</div>
<div id="c2">CSS3</div>
<div id="c3">JAVASCRIPT</div>
```

JS

```
let el = document.getElementsByTagName("div")
el = [...el] //Usando o SPREAD para o transformar em ARRAY
el.map((e,i)=>{
  console.log(e.innerHTML)
})
```

Resultado:

HTML5

CSS3

JAVASCRIPT

- **Método THIS**

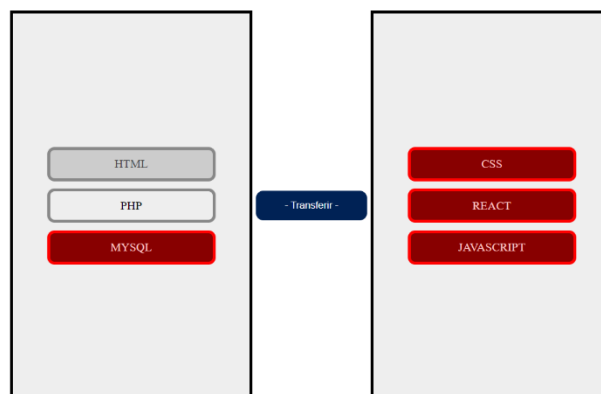
EX:

```
function aluno(nome, nota){
  this.nome = nome // 'this.nome' faz referencia ao um elemento
da minha função chamado 'nome' ('this.nome' é como se fosse
uma variavel declarada para a função aluno, que recebe o valor
do parametro 'nome')
  this.nota = nota
```

```
    console.log(nome)
    console.log(nota)
  }
  aluno("Wellington", 100)
  Resultado: Wellington, 100
```

✓ PRÁTICA Nº 1

Transferindo elementos de uma caixa para a outra.



CÓDIGO

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <title>Praticando addEventListener</title>

  <style>
    *{
      padding: 0;
      margin: 0;
      border: 0;
    }
    main{
      display: flex;
      justify-content: center;
      align-items: center;
      width: 100%;
    }
    button{
      width: 150px;
      height: 40px;
      background: #025;
      color: #fff;
      cursor: pointer;
      border-radius: 10px;
    }
    .curso{
      display: flex;
      justify-content: center;
      width: 200px;
      border: 4px solid #888;
      border-radius: 10px;
      padding: 10px;
      margin: 5px 0px;
      cursor: pointer;
    }
    .selecionado{
      background: #800 !important;
      color: #fcc !important;
      border-color: #f00 !important;
    }
    .c1{
      background-color: #ccc;
      color: #444;
```

```

    }
    .c2{
      background-color: #444;
      color: #ccc;
    }

    .caixa{
      border: 4px solid #000;
      background:#eee;
      padding: 10px;
      display: flex;
      flex-direction: column;
      justify-content: center;
      align-items: center;
      margin: 5px;
      height: 500px;
      width: 300px;
    }

</style>

</head>
<body>

  <main>
    <div id="caixa1" class="caixa">
      <div id="c1" class="curso c1">HTML</div>
      <div id="c2" class="curso c2">CSS</div>
      <div id="c3" class="curso c3">JAVASCRIPT</div>
      <div id="c4" class="curso c4">PHP</div>
      <div id="c5" class="curso c5">REACT</div>
      <div id="c6" class="curso c6">MYSQL</div>
    </div>
    <button id="btn_trasnferir">- Transferir -</button>
    <div id="caixa2" class="caixa">

      </div>
  </main>

  <script>

    const caixa1 = document.querySelector("#caixa1")
    const caixa2 = document.querySelector("#caixa2")
    const btn = document.querySelector("#btn_trasnferir")

    let todosCursos = [...document.querySelectorAll('.curso')]
    console.log(todosCursos)

    todosCursos.map((el)=>{

```

```

        el.addEventListener('click', (evt)=>{
            const curso = evt.target //Pegando o elemento
            curso.classList.toggle("selecionado");//Se tem a classe
            tira, se não tem, adiciona.
        })
    })

    btn.addEventListener('click', ()=>{
        const cursosSelecionados =
[...document.querySelectorAll(".selecionado")]
        const cursosNaoSelecionados =
[...document.querySelectorAll(".curso:not(.selecionado)")]//Todos
elementos que possui a classe curso, traga para mim, os elementos que não
possui a classe 'selecionado'
        console.log(cursosSelecionados)
        console.log(cursosNaoSelecionados)

        cursosSelecionados.map((e)=>{
            caixa2.appendChild(e)
        })

        cursosNaoSelecionados.map((e)=>{
            caixa1.appendChild(e)
        })
    })

</script>

</body>
</html>

```

- **stopPropagation()**

O `stopPropagation()` método da Event interface evita a propagação do evento atual nas fases de captura e borbulhamento. No entanto, isso não impede a ocorrência de qualquer comportamento padrão; por exemplo, os cliques nos links ainda são processados. Se você deseja interromper esses comportamentos, consulte o `preventDefault()` método.

EX:

```
<main>
```

```
  <div id="caixa1" class="caixa">
    <div id="c1" class="curso c1">HTML</div>
    <div id="c2" class="curso c2">CSS</div>
    <div id="c3" class="curso c3">JAVASCRIPT</div>
    <div id="c4" class="curso c4">PHP</div>
    <div id="c5" class="curso c5">REACT</div>
    <div id="c6" class="curso c6">MYSQL</div>
  </div>
```

```
</main>
```

```
<script>
```

```
  const caixa1 = document.querySelector("#caixa1")
```

```
  caixa1.addEventListener('click', (evento)=>{
```

```
    console.log(evento.target)
```

```
    console.log("Clicou")
```

```
  })
```

//Por comum, ao clicar em qualquer lugar dentro da 'caixa', o evento ira propagar.

```
  const cursos = [...document.querySelectorAll(".curso")]
```



```
cursos.map((e)=>{  
  e.addEventListener('click', (evento)=>{  
    evento.stopPropagation()  
  })  
})
```

//Com o uso do 'stopPropagation' nos botões, ao clicar neles, o evento anterior não ira se propagar.

```
</script>
```

- **Entendendo a relação dos elementos no DOM**

```
<main>
```

```
<div id="caixa1" class="caixa">  
  <div id="c1" class="curso c1">HTML</div>  
  <div id="c2" class="curso c2">CSS</div>  
  <div id="c3" class="curso c3">JAVASCRIPT</div>  
  <div id="c4" class="curso c4">PHP</div>  
  <div id="c5" class="curso c5">REACT</div>  
  <div id="c6" class="curso c6">MYSQL</div>  
</div>
```

```
</main>
```

```
<script>
```

```
const caixa1 = document.querySelector("#caixa1")
```

```
const btn_c = [...document.querySelectorAll(".curso")]
```

```
console.log(document.getRootNode()) //Todos elementos
```

```
console.log(caixa1.firstChild) //primeiro elemento da coleção
```

```
console.log(caixa1.lastChild) //Ultimo elemento da coleção
```

```
console.log(caixa1.children) //Pai dos elementos
```

```
console.log(caixa1.hasChildNodes()) //retorna 'true' se o elemento possui um filho
```

```
//Obs: Se houver apenas texto dentro do elemento, ele também irá retornar 'true'
```

```
console.log( btn_c[0].children.length > 0 ? "Possui filhos!" : "Não possui filhos!") //Corrigindo o problema anterior
```

```
console.log(caixa1.children[1].innerHTML = "TESTE") //Mudando o nome do texto do elemento
```

```
/*
```

```
PARA PEGAR O AVÔ DO ELEMENTO, BASTA CHAMAR O  
'parentNode.parentNode'.
```

EX:

```
<div id="avo">
```

```
  <div id="pai">
```

```
    <div id="filho">
```

```
      HELOO!
```

```
    </div>
```

```
  </div>
```

```
</div>
```

```
const filho = document.querySelector("#filho")
```

```
console.log(filho.parentNode.parentNode)
```

```
*/
```

```
</script>
```

- **Adicionando Elementos no DOM**

```
<main>
```

```
  <div id="caixa1" class="caixa">
```

```
    <!-- Cursos serão adicionados aqui -->
```

```
  </div>
```

```
</main>
```

```
<script>
```

// FORMA AUTOMATICA

```
const caixa1 = document.querySelector("#caixa1")
const cursos = ["HTML", "CSS", "JAVASCRIPT", "PHP", "REACT", "MYSQL"]

cursos.map((e, chave) => {
    const novoElemento = document.createElement("div")
    novoElemento.setAttribute("id", "c" + (chave + 1))
    novoElemento.setAttribute("class", "curso c" + (chave + 1))
    novoElemento.innerHTML = e
    caixa1.appendChild(novoElemento)
})
```

// FORMA MANUAL

```
const caixa1 = document.querySelector("#caixa1")

const novoElemento = document.createElement("div") //Criando 'div'
novoElemento.setAttribute("id", "c7") //Criando um 'id' com o
nomen'c7'
novoElemento.setAttribute("class", "curso c7") //Criando uma 'class'
com o nome 'curso e c1'
novoElemento.innerHTML = "ReactNative" //Criando texto no
elemento
caixa1.appendChild(novoElemento) //Adicionando esse elemento
como o filho do 'caixa1'
```

</script>

- **Removendo Elemento do DOM**

EX:

<main>

```
<div id="caixa1" class="caixa">
  <div id="c1" class="curso c1">HTML</div>
  <div id="c2" class="curso c2">CSS</div>
  <div id="c3" class="curso c3">JAVASCRIPT</div>
  <div id="c4" class="curso c4">PHP</div>
  <div id="c5" class="curso c5">REACT</div>
  <div id="c6" class="curso c6">MYSQL</div>
</div>
```

</main>

<script>

```
const caixa1 = document.querySelector("#caixa1")
const btn_c = [...document.querySelectorAll(".curso")]
```

```
btn_c.map((e)=>{
  e.addEventListener('click', (evt)=>{
```

```
        console.log(evt.target)
        caixa1.removeChild(evt.target) // Removendo um filho do
        elemento 'caixa1'
    })
})

</script>
```

- **método FILTER**

O **filter()** método cria um novo array preenchido com elementos que passam em um teste fornecido por uma função.

```
const idades = [15, 33, 16, 18, 50, 12, 8]
console.log(idades)
const maior = idades.filter((valor, indice, array)=>{
    if(valor >= 18){
        return valor
    }
})
console.log(maior)
```

Resultado:

[33, 18, 50]

//(VALOR) Valor

//(INDICE) Posição que está no array

```
//(ARRAY) Retorna todo array
```

- **Método Toggle**

Altere entre adicionar e remover um nome de classe de um elemento com JavaScript.

EX:

```
<style>
```

```
  #caixa{  
    width: 300px;  
    height: 300px;  
    border: solid 2px black;  
  }
```

```
  .teste{  
    background: red;  
  }
```

```
</style>
```

```
<div id="caixa"> //Aqui será adicionado o background com o toggle
```

```
</div>
```

```
<script>
```

```
  document.getElementById('caixa').addEventListener('click',  
    ()=>{
```

```
document.getElementById('caixa').classList.toggle('teste')
//Se o elemento 'caixa' não tiver a class 'teste', ele
adiciona, se tiver, ele tira.

})
</script>
```

- **Método FIND**

O find() método retorna o primeiro elemento na matriz fornecida que satisfaça a função de teste fornecida. Se nenhum valor satisfizer a função de teste, undefined é retornado.

EX:

```
const array1 = [5, 12, 8, 130, 44];
const found = array1.find((elem, indic, array){
    if(elem == 8){
        return elem
    }
});
console.log(found);
Resultado: 8
```

//O método 'find' faz com que a função percorra todo o array, e quando for true, ela para a execução e retorna o argumento.

- **método EVERY**

O método `every()` testa todos os elementos do array passam pelo teste de implementação pela função fornecida. Este método retorna um valor booleano.

EX:

```
const array1 = [1, 30, 39, 29, 10, 13];  
array1.every((elem, indic, array)=> {  
    return elem > 18  
})
```

//Vai retornar true se todos elementos do array forem maior que 18, nesse caso ira retornar false.

- **método SOME**

O método `some()` testa se ao menos um dos elementos no array passa no teste implementado pela função atribuída e retorna um valor true ou false.

EX:

```
const array1 = [1, 5, 18, 8, 10, 13];  
array1.some((elem, indic, array)=> {  
    return elem >= 18  
})
```

```
}}
```

//O método 'some' vai retornar true se ao menos um elemento do array passar no teste implementado pela função.

- **Método REDUCE()**

O método `reduce()` executa uma função redutora (fornecida por você) para cada elemento do array, gerado em um único valor de retorno.(Gera uma redução do array)

EX:

Somando o valor atual com o anterior.

```
<script>
```

```
const resultado = document.getElementById('resultado')
```

```
const elementosArray = [1,2,3,4,5]
```

```
let anti = []
```

```
let atu = []
```

```
let dobro = []
```

```
elementosArray.reduce((anterior, atual, pos)=>{
```

```
  anti.push(anterior)
```

```
  atu.push(atual)
```

```
  dobro.push(atual * 2)
```

```
  return atual + anterior
```

```
    })  
    resultado.innerHTML += "<br />Valor anterior:" + anti + "<br  
/>Valor Atual:" + atu + "<br />Dobro do valor atual: " + dobro  
  })  
</script>
```

Resultado:

Soma do valor anterior com o atual: 15

Valor anterior:1,3,6,10

Valor Atual:2,3,4,5

Dobro do valor atual: 4,6,8,10

- **ITERADORES**

Um objeto é um iterador (iterador) quando sabe como acessar itens numa coleção, um por vez, enquanto mantém rastreada a posição atual em uma sequência de dados. Em JavaScript um iterador é um objeto que oferece o método `next()`, o qual retorna o próximo item da sequência. Este método retorna um objeto com duas propriedades: `done` e `value`.

EX:

```
<script>  
  const valores = [10, 8, 9, 2]  
  const it_valores = valores[Symbol.iterator]()  
  
  console.log(valores)  
  console.log(it_valores.next())
```

```
console.log(it_valores.next())  
console.log(it_valores.next())  
console.log(it_valores.next())  
console.log(it_valores.next())
```

//Usando em strings

```
const texto = 'hello'  
const it_text = texto[Symbol.iterator]()
```

```
console.log(texto)  
console.log(it_text.next())  
console.log(it_text.next())  
console.log(it_text.next())  
console.log(it_text.next())  
console.log(it_text.next())  
console.log(it_text.next())  
</script>
```

Resultado:

(4) [10, 8, 9, 2]

{value: 10, done: false}

{value: 8, done: false}

{value: 9, done: false}

{value: 2, done: false}

{value: undefined, done: true}

hello

{value: 'h', done: false}

{value: 'e', done: false}

{value: 'l', done: false}

{value: 'l', done: false}

{value: 'o', done: false}

{value: undefined, done: true}