

Novo Curso de Javascript Completo, Profissional e Moderno - Curso de Javascript Moderno - Aula 01

Javascript é uma linguagem que roda tanto do lado do cliente quanto do lado do servidor.

Devemos baixar o node, podemos instalar o nodeJs no próprio site deles.

Para conferir se o nodeJs está instalado podemos escrever no prompt de comando **Node -v**

Para começar a escrever nosso código em Javascript devemos criar um arquivo com .js no final do nome, o nome pode ser qualquer um por exemplo teste.js desde que tenha .js no final.

Javascript é uma linguagem interpretada e não compilada significa que é executada em tempo real não preciso compilar o código e criar um executável.

Console.log

Faz a impressão no console, e o comando de saída, tudo que for escrito nele é imprimido no console.

```
console.log ("Alo, mundo!");
```

Esse código vai imprimir no console o texto Alo, mundo!

Comentários

Os comentários em um código podem servir para diversas coisas como por exemplo explicar para que serve um código ou até mesmo organizar, os comentários não serão rodados ou executados no código e apenas algo visual para quem está escrevendo no código, resumindo não vai mudar nada no resultado.

Para fazer um comentário em js existe duas maneiras, se fizermos o comentário em uma linha somente nos colocamos duas barras (//) e o texto logo em seguida,

```
// Comentário de uma linha
```

mas se quisermos comentar um bloco de código com várias linhas podemos colocar o código entre uma aspa e um asterisco (/* comentário aqui */).

```
/*  
  Esse  
  é um comentário  
  Em Js  
*/
```

Como rodar Javascript dentro do browser?

Para rodar o Javascript no browser devemos unir o Javascript com o html, então devemos criar um arquivo .html, e chamar o Javascript dentro do HTML, depois de criar o arquivo .html devemos escrever o código base do html

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <title>Aula 1 de JavaScript-CFBCursos</title>
</head>
<body>

</body>
</html>
```

E dentro da área de body que é o corpo do site escrevemos

```
<script src="aula01.js"></script>
```

Aqui nós chamamos o arquivo .js para dentro do html e quando executamos o html e aberto o navegador junto com o Javascript que já está sendo executado.

É importante que coloquemos a tag script dentro do body e no final de todo código que é escrito dentro do body, pois quando a gente for escrever DOM que é a manipulação de elementos em Javascript logo veremos isso, e preciso que o html seja carregado antes do Js.

Javascript em Modo Estrito, você sabe o que é? - Curso de Javascript Moderno - Aula 02

Modo estrito

Serve para deixar nosso código mais limpo com menos sujeira, com menos coisas inutilizadas, por exemplo no modo estrito não conseguimos usar uma variável que não foi declarada, e as variáveis criadas tem que ser usadas senão do erro, então erros que poderiam ser executados sem o modo estrito passam a ser necessários correção, o modo estrito não deixa passar nenhuma falha no código.

Para ativar o modo estrito nos colocamos no código

```
"use strict"
```

Que já será ativado,

Diferença entre LET, VAR e CONST em Javascript - Curso de Javascript Moderno - Aula 03

O que é uma variável

É uma posição dentro da memória ram, quando nós estamos criando uma variável nós estamos basicamente criando um espaço dentro da memória ram e nomeamos esse espaço onde podemos colocar algum dado dentro desse espaço.

Pense em um armário com 8 gavetas onde temos as gavetas enumeradas tipo gaveta 1, gaveta 2 e por aí em diante, cada gaveta dessa é um espaço no armário onde podemos guardar materiais e a mesa lógica da variável onde o armário é a memória ram em geral e as gavetas são as variáveis que são espaços vazios para guardar coisas, e os materiais são os dados que serão guardados.

Declarando uma variável

```
var nome = "John"
```

Esta é uma variável, primeiro declaramos a variável usando VAR e o nome do espaço na nossa memória Ram (variável) se chama nome e depois utilizamos o sinal de = (igual) que no Javascript tem o sentido de atribuir então a palavra "John" (string) e colocada dentro da variável nome.

Outra coisa é que para usar a variável ela deve estar acima no código para podermos utilizar lá em baixo, então primeiro ela precisa ser declarada para depois utilizarmos pois na hora da execução de código, o navegador lê o código de cima para baixo, então você concorda que precisa existir a variável antes de podermos usar ela.

E se por exemplo quisermos usar a variável lá no final do código ou em qualquer lugar, a variável não pode ser declarada dentro por exemplo de uma function, ou em uma condicional que são fechadas por {} (chaves), pois a variável será declarada somente lá dentro e para usar a variável fora das chaves é impossível pois lá ela não existe.

Podemos declarar uma variável de 3 maneiras

1 maneira

```
let num1=10  
let num2=10
```

2 maneiras

```
let num1=10, let num2=10
```

Dessa maneira poderemos declarar várias variáveis em uma linha so separadas por vírgulas.

3 maneiras

```
let num1=num2=num3=10
```

Aqui declaramos 3 variáveis com o valor 10 sendo que, a variável recebe o número 10 aí a variável num2 recebe o valor da variável num3 e a variável num1 recebe o valor da variável num2.

Diferença entre Let, Const e Var

Resumido declarando a variável com **let** se colocarmos dentro de uma condicional por exemplo a variável só poderá ser usada lá dentro e não fora, mas agora se usar Var ela poderá ser usada somente dentro e fora dessa condicional, mas agora se a condicional estiver dentro de uma function o var e o let não poderão ser usados fora da function.

Var = “Se for declarada dentro de um escopo global poderá ser utilizado por todo o escopo, mas se declarada dentro de um escopo local, poderá ser utilizado apenas nele.”

Let = “Se for declarada dentro de um escopo global poderá ser utilizado por todo o escopo, e se for declarada dentro de um escopo local poderá ser utilizado dentro dele todo e uma camada fora dele, mas se um escopo local estiver dentro de um escopo local ele poderia sair de apenas um escopo e será utilizado lá dentro também.”

Const = “Quando a variável for declarada com algum dado, o valor dessa variável depois não poderá ser mudado, o único dado que vai ficar é o que foi atribuído quando a variável foi declarada, resumindo não podemos atribuir um novo valor na variável declarada com const”

Mudança entre tipo de dados em variáveis e no conteúdo

Podemos mudar algo que está dentro da variável apenas atribuindo seu respectivo dado, então se já declaramos a variável antes com algum valor, se quisermos mudar o conteúdo dessa variável podemos apenas atribuir um outro valor a ela.

```
let nome = "JohnChk"  
nome = 10  
console.log(nome)
```

O resultado que será imprimido vai ser o número 10, pois ele foi atribuído depois da criação da variável e uma variável cabe apenas um dado em seu espaço.

Aprenda sobre os Operadores Matemáticos em Javascript - Curso de Javascript Moderno - Aula 04

Operadores aritméticos

`+` (Operador de soma)

`-` (subtração)

`/` (divisão)

`*` (multiplicação)

`%` (resto de divisão)

`++` (incremento)

`--` (decremento)

`+=` (incremento)

`-=` (decremento)

A precedência de operadores conta a mesma regra da matemática tradicional.

.

Operações exemplos

```
Num1=5  
num2=10  
  
res=num1+num2  
console.log(res)
```

Aqui a variável num1 tem o valor 5 e a variável num2 tem o valor 10, aí pegamos a variável res e jogamos dentro dela o resultado da soma de num1 + num2 que será 15.

.

Podemos fazer também dentro do console.log()

```
num1=5  
num2=10  
console.log(num1-num2)
```

Aqui nos subtraímos num1-num2 e será mostrado a resposta no terminal que e onde o console.log mostra quando imprime o resultado será 5.

.

Ordem de precedência

```
num1=5  
num2=10  
console.log(num1-num2)*2
```

nesse caso pensando na ordem de precedência primeiro será num1-num2 por causa dos parênteses, e depois o resultado será multiplicado por 2 assim mostrando o valor deste calculo que e 10.

.

Resto da divisão

```
num1=15  
num2=2  
res=num1%num2  
console.log(res)
```

Aqui estamos dividindo com % que na verdade não mostrara o resultado da divisão, mas sim o resto dessa divisão, entao 15 dividido por 2 da 7 mas resta 1, oque será mostrado no console.log será 1 pois é o resto.

.

Incremento

```
num1=10  
num1++  
num1++
```

Aqui nesse caso nossa variável num1 tem o valor 10, mas o incremento sempre adiciona um numero a essa variável entao quando colocamos o nome da variável que é num1 e colocamos ++ será adicionado um numero, entao como fizemos 2 vezes o valor final da variável num1 será 12.

.

decremento

```
num1=10  
num1--  
num1--
```

Aqui nesse caso nossa variável num1 tem o valor 10, mas o decremento sempre retira um número dessa variável entao quando colocamos o nome da variável que é num1 e colocamos -- será retirado um numero, entao como fizemos 2 vezes o valor final da variável num1 será 8.

.

Incremento PRO

Tem a mesma função do incremento so que podemos escolher o valor que queremos incrementar, quanto no incremento comum podemos somar apenas 1 a variável aqui no podemos escolher quanto queremos incrementar.

```
num1 = 10;  
num1+= 5;
```

neste exemplo será incrementado o valor 5 a variável num1 ficando assim 15, ele estará adicionando mais um valor, podemos incrementar outras variáveis com algum valor

```
num1 = 10;  
num2 = 3  
num1+= num2;
```

O resultado será 13 pois o valor de num1 era 10 aí foi incrementado o valor de num2 ficando assim o resultado final 13.

.

Decremento PRO

Tem a mesma função do decremento comum so que podemos escolher o valor que queremos decrementar, enquanto no decremento comum podemos subtrair apenas 1 a variável aqui nós podemos escolher quanto queremos decrementar.

```
num1 = 10;  
num1-= 5;
```

neste exemplo será decrementado o valor 5 a variável num1 ficando assim 5, ele estará retirando mais um valor, podemos tambem decrementar outras variáveis com algum valor

```
num1 = 10;  
num2 = 3  
num1-= num2;
```

O resultado será 7 pois o valor de num1 era 10 aí foi decrementado o valor de num2 ficando assim o resultado final 7.

.

Operadores Relacionais em Javascript, aprenda como usar! - Curso de Javascript Moderno - Aula 05

O que é uma operação relacional?

Operação relacional é uma comparação usaremos operadores de comparação

Operadores

> Maior

```
let num1=10
let num2=5
let num3=10
console.log(num1 > num2)
```

Aqui estamos perguntando no console.log se num1 que tem 10 atribuído a ele é maior que num2 que tem o número 5 atribuído a ele, o resultado será verdadeiro (true) pois num1 é maior que num2.

.

>= Maior ou igual

```
let num1=10
let num2=5
let num3=10
console.log(num1 >= num3)
```

Aqui estão perguntando se num 1 é maior ou igual a num3, então se for maior ou igual o resultado será verdadeiro(true) senão se não for o resultado será falso(false).

O resultado será verdadeiro(true) pois num1 é igual a num3 mesmo que não seja maior, pode ser qualquer uma das opções, se for maior ou igual.

.

< Menor

```
let num1=10
let num2=5
let num3=10
console.log(num1 < num2)
```

Aqui estamos perguntando no console.log se num1 que tem 10 atribuído é menor que num2 que tem o número 5 atribuído a ele, o resultado será falso (false) pois num1 não é menor que num2.

.

<= Menor ou igual

```
let num1=10, let num2=5, let num3=10
console.log(num1 <= num3)
```

Aqui estão perguntando se num1 é menor ou igual a num3, então se for menor ou igual o resultado será verdadeiro(true) senão se não for o resultado será falso(false).

O resultado será verdadeiro(true) pois num1 é igual a num3 mesmo que não seja menor, pode ser qualquer uma das opções, se for menor ou igual.

.

== igual

```
let num1=10, let num2=5, let num3=10
console.log(num1 == num3)
```

Aqui estamos perguntando se num1 é igual a num3, o resultado será verdadeiro (true) pois num1 é igual a num3.

.

= Atribuição

Atribuição nós já vimos antes e já usamos, usamos ao atribuir um valor(dado) a uma variável.

```
let num1=10
```

Neste exemplo estamos atribuindo o valor(dado) número 10 a variável nomeada como num1.

.

!() not

Esse é um pouco diferente dos demais pois ele inverte os valores

```
let num1=10, let num3=10
console.log(num1 == num3)
```

por exemplo nesta comparação o valor será verdadeiro(true) pois num1 é igual a num3, mas aí nós adicionamos o sinal NOT a ele.

```
console.log(!(num1 == num3))
```

nós adicionamos e colocamos os valores entre colchetes, o sinal de NOT(!()) ele nega, então se o valor deu verdadeiro(true), será inverso para false, e se o valor der false será invertido para true, ele simplesmente inverte o resultado.

.

!=

```
let num1=10
let num2=5
let num3=10
console.log(num1 != num3)
```

Aqui estamos perguntando se num1 é diferente (!=) de num3 e o resultado é obviamente falso(false), pois num1 e num2 têm o mesmo valor.

.

Respondendo Perguntas dos Inscritos

Parte 1 (aulas de 1 a 5) - Curso de Javascript Moderno - Aula 06

Qual a diferença entre == e ===

O == faz comparação do valor da variável e apenas isso, uma comparação rasa.

O === Faz uma comparação mais profundo pegando também o tipo de dados, se é uma string, um número entre outras coisas, então além de comparar os valores ele compara o tipo de dados e também o lugar na memória.

Temos também a seguinte questão

```
let v1={nome:"Bruno"}  
let v2={nome:"Bruno"}
```

```
console.log(v1===v2)
```

Por mais que os valores sejam o mesmo e a variável tenha o mesmo tipo, eles não ocupam o mesmo lugar na memória então não são iguais.

O nome da variável não importa

.

Trabalhando pelo prompt de comando

Podemos fazer isso colocando o caminho da nossa pasta no cmd, mas temos que verificar que diretório estamos usando pode ser que seja o c ou o d, caso nossa pasta esteja no diretório d nos apenas colocamos d: se nossa pasta com o projeto estiver no diretório c nos colocamos c: e depois escrevemos node (nome do arquivo) sem os parênteses.

.

Diferença entre Diferente e NOT

O sinal de diferente != ele confere se algo é diferente do outro, se for diferente da verdadeiro(true pois é uma verdade mas o NOT ele simplesmente pega a comparação e inverte o resultado, então se uma comparação der verdadeiro o NOT converte para falso(false).

.

Codigo em JS para descobrir em qual dispositivos estamos usando na pagina web

```
if(navigator.userAgent.match(/Android/i)
|| navigator.userAgent.match(/WebOS/i)
|| navigator.userAgent.match(/iPhone|iPad|iPod/i)
|| navigator.userAgent.match(/BlackBerry/i)
|| navigator.userAgent.match(/Windows Phone/i)
|| navigator.userAgent.match(/Opera Mini/i)
|| navigator.userAgent.match(/IEMobile/i)
){
    console.log("Celular")
}else{
    console.log("PC")
}
```

Oque eo DOM (manipulação de dados)

E a arvore de elementos dentro do html, temos uma estrutura que são os elementos da nossa pagina, o DOM ele manipula os elementos identificando a arvore de elementos no navegador.

Ele cria essa arvore de elementos (DOM) e possibilita que o JavaScript o manipule.

Defer

Quando colocamos a propriedade defer

```
<script src="aula06.js" defer></script>
```

Ele faz com que de uma preferencia para que o HTML seja carregado primeiro para depois o JavaScript seja lido, entao podemos deixar esse codigo para chamar o javaScript la no header invece de la embaixo como e normalmente posto, o resultado eo mesmo, mas a prefrenca e que seja posto la em baixo ainda.

Operadores Lógicos em Javascript, aprendendo a usar! - Curso de Javascript Moderno - Aula 07

&& -> and -> e

|| -> or -> ou

!() -> not -> não

AND		
V	V	V
V	F	F
F	V	F
F	F	F

OR		
V	V	V
V	F	V
F	V	V
F	F	F

AND		
1	1	1
1	0	0
0	1	0
0	0	0

OR		
1	1	1
1	0	1
0	1	1
0	0	0

Essa tabela nos ajudara,

&& (E lógico):

O operador **&&** é usado para verificar se duas condições são ambas verdadeiras. Ele retorna true somente se ambos os lados da expressão forem avaliados como verdadeiros. Se pelo menos um dos lados for falso, o operador **&&** retorna false.

```
let a = true;
let b = false;
console.log(a && b); // Retorna false, porque a é verdadeiro mas b é falso
console.log(a && true); // Retorna true, porque ambos a e true são verdadeiros
```

|| (OU lógico):

O operador || é usado para verificar se pelo menos uma das condições é verdadeira. Ele retorna true se pelo menos um dos lados da expressão for avaliado como verdadeiro. Somente quando ambos os lados forem falsos, o operador || retorna false.

```
let a = true;
let b = false;
console.log(a || b); // Retorna true, porque a é verdadeiro, mesmo que b
// seja falso
console.log(false || false); // Retorna false, porque ambos os lados são
// falsos
```

! (Negação lógica):

O operador ! é usado para negar o valor de uma expressão booleana. Ele inverte o valor de verdadeiro para falso e de falso para verdadeiro.

```
let a = true;
console.log(!a); // Retorna false, porque a é verdadeiro e a negação é
// falsa
let b = false;
console.log(!b); // Retorna true, porque b é falso e a negação é
// verdadeira
```

Operadores Bitwise em Javascript - Curso de Javascript Moderno - Aula 08

AND			OR		
V	V	V	V	V	V
V	F	F	V	F	V
F	V	F	F	V	V
F	F	F	F	F	F

AND			OR		
1	1	1	1	1	1
1	0	0	1	0	1
0	1	0	0	1	1
0	0	0	0	0	0

O BitWise nos operamos os bits.

E um pouco complicado de entender, usaremos o numero 10 e 11

```
let n1=10; /*10 em binario (1010) */
let n2=11; /*11 em binario (1011) */

let res = n1 & n2

console.log(res) /* resultado sera 10 */
```

temos o codigo acima e o resultado acima sera 10, mas porque?

Se pegarmos os códigos binários as casas dos bits que são equivalentes eles irão se repetir por causa do **&**, pois como já vimos anteriormente o **&** exige que os dois seja equivalentes, então pegamos o primeiro numero em binário do 10 e do 11,

(1010)

(1011)

O primeiro numero de ambos e 1 e eles se repetem, então o retorno sera 1, o segundo numero de ambos e 0 e se repetem então o resultado sera 0, o terceiro numero de ambos e 1 então o retorno sera 1, o 4 numero de ambos eles não se repetem pois temos o numero 1 e o numero 0 então o retorno sera 0 (quando não se repete o retorno sera 0) ficando assim (1010).

E o codigo em binário do numero 10 e (1010) então por isso o resultado la no nosso codigo foi 10.

```
let res = n1 & n2

console.log(res) /* resultado sera 10 */
```

mas agora fazendo o mesmo exemplo com || (ou)

```
let n1=10; /*10 em binario (1010) */
let n2=11; /*11 em binario (1011) */

let res = n1 || n2

console.log(res) /* resultado sera 11 */
```

temos o código acima e o resultado acima será 11, mas porque?

Se pegarmos os códigos binários as casas dos bits o || falará que pode ser um ou outro os dois não precisam ser equivalentes igual o &, pois como já vimos anteriormente o || não exige que os dois sejam equivalentes mas pode ser um ou outro, então pegamos o primeiro número em binário do 10 e do 11,

(1010)

(1011)

O primeiro número de ambos é 1 e eles se repetem mas tanto faz que pode se repetir ou não, então o retorno será 1, o segundo número de ambos é 0 e se repetem então o resultado será 0, o terceiro número de ambos é 1 então o retorno será 1, o 4º número de ambos eles não se repetem pois temos o número 1 e o número 0 mas o || não se importa porque pode ser um ou outro então o retorno será 1 (quando não se repete o retorno será 1) ficando assim (1011).

E o código em binário do número 11 é (1011) então por isso o resultado lá no nosso código foi 11.

```
let res = n1 || n2

console.log(res) /* resultado sera 11 */
```

mas agora fazendo o mesmo exemplo com ^ (^)

```
let n1=10; /*10 em binario (1010) */
let n2=11; /*11 em binario (1011) */

let res = n1 ^ n2

console.log(res) /* resultado sera 1 */
```

Nesse caso é o mais simples, ele retorna 0 aos bits que se repetem e 1 aos que não se repetem

(1010) -> 10

(1011) -> 11

No nosso exemplo aqui os três primeiros se repetem então o resultado será 0 e o último não se repete então o resultado final é 1.

Operações de deslocamento de BIT

Cara so falar que isso e incrível.

Lembrando os capítulos passados nos vimos sobre os binários e como eles se comportam diante de operadores lógicos, mas agora aprenderemos sobre deslocação de bit, e pegar um bit e modificar ele em JavaScript para nos trazer outro resultado.

BINARIO	NUMERO
1010	10
1011	11

Mostrando em uma planilha do excel podemos ver dois números e seus respecticvos códigos em binário, mas oque aconteceria se deslocássemos o codigo em biinario uma casa para a esquerda, você sabe?

Isso simplesmente faria o codigo binário pular uma casa para a esquerda e o numero 0 preencheria o local vazio ficando assim

BINARIO	NUMERO
10100	20

Nos pulamos uma casa para a esquerda e sempre que um lugar fica vazio e preenchido com 0.

```
let n1=10; /*10 em binario (1010) */
let res = n1 << 1
console.log(res) /*0 resultado sera 20*/
```

Em JavaScript o codigo ficaria assim, pulamos 1 casa para esquerda o numero 10 que esta dentro da variável n1 e depois jogamos o resultado dentro da variável res e mostramos no console, o resultado assim ficando 20 pois o codigo em binário de 20 e (10100).

Mas podemos deslocar para a direita tambem.

```
let n1=10; /*10 em binario (1010) */
let res = n1 >> 1
console.log(res) /*0 resultado sera 5*/
```

Tem a mesma logica do deslocamento para a esquerda so que o codigo agora sera (101_ que da o numero 5 pois foi deslocado o codigo em binário (1010) para a direita e quando deslocamos estamos apagando o ultimo zero.

IMPORTANTE: o deslocamento com apenas 1 numero e apenas um exemplo podemos deslocar 2 casas ou ate mais tambem tanto para a direita quanto para a esquerda resultando em um novo numero.

Onde poderemos usar isto?

Caso você não tenha percebido, sempre que estamos deslocando um bit de algum codigo para a esquerda o numero dobra, e a mesma coisa de multiplicar por 2, entao poderemos usar desta maneira, a mesma coisa deslocando para a direita.

Diferença entre Pré Incremento e Pós Incremento - Curso de Javascript Moderno - Aula 09

O incremento no JavaScript é uma maneira de adicionar um número a uma variável, e como se fosse `num = num + 1`, então se por exemplo temos uma variável com o número 10 e nós fazemos um incremento, logo ele será 11 e se depois incrementarmos de novo ele será 12

Abaixo temos um exemplo de código.

```
let n=10
n++
console.log(n)
```

No código acima tivemos um exemplo de incremento onde o resultado será 11, mas o incremento pode se dividir em duas opções que são pré incremento e o pós incremento

Pré Incremento abaixo:

```
let n=10
console.log(n++)
```

Ele primeiro mostra o número que já está na variável e depois incrementa um número a ela, o que seria imprimido primeiro seria o número 10

pós incremento abaixo:

```
let n=10
console.log(++n)
```

Ela primeiro incrementa um número a variável e depois mostra o valor do número atual, o que seria imprimido primeiro seria o número 11.

.

Decremento

O decremento é a mesma coisa do incremento, só que invés de somar ele vai subtrair.

.

Operador de inversão de sinais

```
let n=10
let x=-n
console.log(x)
```

Podemos converter os sinais com o código acima, o resultado imprimido será -10.

.

concatenação

```
let n1 = 10;
let n2 = 20;
console.log(n1 + n2); /*Mostra o valor da soma entre n1 e n2 */
console.log(n1 + "" + n2); /* Faz uma concatenação inves de somar */
```

no exemplo abaixo serão somadas as 2 variáveis e será mostrada o resultado da soma.

```
console.log(n1 + n2); /*Mostra o valor da soma entre n1 e n2 */
```

Neste exemplo abaixo invés de fazer uma soma iremos na verdade uma concatenação o que será mostrada vai ser "10 vs 20".

```
console.log(n1 + "vs" + n2); /* Faz uma concatenação inves de somar */
```

Aprendendo sobre operador ternário - Curso de Javascript Moderno - Aula 10

É uma forma de reduzir ou simplificar uma operação com uma condicional, podemos utilizar operadores relacionais e operadores lógicos para mostrar esse resultado de uma forma melhor do que true e false.

Condicional

```
let num=11
let res=num%2
if(res==0){
  console.log(`par`)
}else{
  console.log("impar")
}
```

Aqui temos uma operação em condicional, temos a variável num com 11 e a variável res com num%2 que é o resto da divisão entre num e 2. Então o resto de 11 dividido por 2 vai ser igual a 1.

Se(resto da divisão entre num e 2 for igual a 0){

 mostrarNaTela(`Par`)

}senão{

 Mostrar(`Impar`)

}

Então resumindo se 0 o resultado do cálculo em res for igual a 0 deverá ser mostrando par, senão(else) ou caso não seja igual a 0 será mostrado impar.

Operador ternário

```
// Teste logico ? se verdadeiro(1) : se false(0)
```

Abaixo temos um exemplo de operador ternário

```
rest=(2+2==4 ? "true(1)" : "false(0))"  
console.log(rest)
```

Aqui já no início do código vemos que rest vai pegar o resultado de todo esse cálculo

E o cálculo é o seguinte: se 2+2 for igual a 4 será mostrado true(0) senão se for falso será mostrado false(0).

O resultado foi true(0) pois é uma verdade.

Devemos lembrar que

True é igual a 1 e false é igual a 0.

Outro exemplo

Em vez de colocar todo aquele código condicional, nós usamos apenas

```
let num=10  
res=(num%2 ? "Par" : "Impar")  
console.log(res)  
//0 = False  
//1 = True
```

Nesse código temos uma variável chamada num com o número 10, mas abaixo aqui já simplificamos um pouco as coisas

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

Deveremos lembrar pela explicação abaixo que o resto de num%2 acima é igual a 0

Esse cálculo acima é quase a mesma coisa do código condicional que já fizemos lá em cima, mas logo você verá que teremos um problema, continue lendo esse código, pegue o resultado de num%2 e se for igual a 1 será imprimido par, se for igual a 0 será imprimido Impar, mas o porquê disso? Se você reparar bem, estamos com valores invertidos, pois o restante de zero deve dar par e não impar, isso é porque

Devemos lembrar primeiramente que false(falso) é igual a 0, e true(verdadeiro) é igual a 1.

```
//0 = False  
//1 = True
```

Essa é uma regra básica da programação

```
? "Par" : "Impar"
```

Mas o problema é que o resultado deu 0, e 0 é igual a false e quando é false será mostrado a segunda opção, que neste caso é "impar", mas nós queremos que quando for 0 que seja mostrado par e não impar, então resumindo esse cálculo, está apresentando o resultado de forma errada, pois 10%2 o restante é 0, mas é igual a false, mostrando assim a segunda opção que é impar.

Faremos seguinte, então

```
let num=10
res=(!(num%2) ? "Par" : "Impar")
console.log(res)
```

Pronto resolvemos colocando uma negação então se o resultado de `num%2` for igual a 0 que é false será invertido para true, pois estamos usando uma negação no código, e com isso resolvemos

.

Podemos ter a seguinte questão vamos supor que eu esteja recebendo dados de um banco de dados, e criamos uma variável onde ela recebe o status de um cliente onde se ele for ativo ele envia a letra a e se for inativo o banco de dados envia a letra i

```
let st="a"
res=(st == "a" ? "Ativo" : "Inativo")
console.log(res)
```

Aqui fizemos o seguinte se o status do cliente for igual a "a" de ativo será mostrado "ativo" mas caso seja diferente de "a" que provavelmente será i pois só temos essas 2 opções no banco de dados será mostrado "inativo".

.

Operador Typeof, retornando o tipo da variável - Curso de Javascript Moderno - Aula 11

Com o `Typeof` conseguimos saber com exatidão o tipo de variável que estamos lidando.

Abaixo temos os seguintes tipos de dados

```
let v1=10//Numero
let v2="10"//String
let v3=v1===v2//boolean
let v4={nome:"Bruno"}//Objeto
```

v1 = São todos os números

v2 = São as strings que são formatos de textos são mostrados o que realmente estão escritos

```
let v2="10 + 10"//String
```

se fosse um número por exemplo o resultado imprimido no `console.log()` seria 20, mas como é uma string ele mostra exatamente o que está escrito então o resultado é 10 + 10.

V3 = Esse é o tipo Booleano que é usado de comparação para compararmos se é igual, maior ou menor < e etc.

V4 = Esse tipo se chama objeto, mais pra frente veremos sobre ele.

.

typeof

Mas não precisamos decorar o tipo de dados pois podemos ver qual é ele com um código e as vezes precisamos de saber qual o tipo de um certo dado no código para podermos por exemplo converter para outro tipo de dado

No código abaixo usamos o `typeof` para descobrir o tipo de dados da variável `v2`, e descobrimos que o tipo dela é `string`

```
let v2="10 + 10">//String
console.log(typeof(v2))
```

e será imprimido o tipo de dado, não precisa ser só no `console.log` podemos por exemplo jogar o resultado em uma variável

```
res=typeof(v2)
```

Ai dentro de `res` estará escrito `string` pois foi jogado dentro de `res` o tipo de dado de `v2`.

Operador Spread, aprendendo sobre o espalhador em Javascript - Curso de Javascript Moderno - Aula 12

Ele quebra um array, e vai devolver elemento a elemento.

Vai simplificar muito quando se trata de array

Nos podemos concatenar objetos dentro do código

```
const jogador1={nome:"Bruno",energia:100,vidas:3, magia:150};
const jogador2={nome:"Bruce",energia:100,vidas:5, velocidade:80};
const jogador3={...jogador1,...jogador2}
console.log(jogador3)
```

Se por exemplo temos o objeto `jogador1` com seus dados, e o objeto `jogador2` com seus dados, depois nós pegamos e usamos `spread` no `jogador3` criando tipo uma junção do `jogador1` com o `jogador2` formando assim o `jogador3`, o resultado será

```
{ nome: 'Bruce', energia: 100, vidas: 5, magia: 150, velocidade: 80 }
```

Como os dois já tinham `nome`: foi usado somente o do último, `energia` os dois têm então foi mantido e `vida` foi utilizada a do último colocado, mas agora `magia` e `velocidade` é uma característica de cada um e foi pego e colocado no `jogador3`.

Podemos usar o spread da seguinte maneira,

```
const soma=(v1,v2,v3)=>{  
  return v1+v2+v3  
}  
let valores=[1, 5, 4]  
console.log(soma(valores))
```

Aqui temos um array function que faz a soma do valor v1, v2, v3 e la embaixo no console.log nos chamamos essa function, so que os dados nos pegamos dentro da array

```
let valores=[1, 5, 4]
```

mas quando vamos executar vemos que da o seguinte erro

```
1,5,4undefinedundefined
```

Ele além de não somar ficou todo bagunçado, oque acontece e que na hora da function pegar os números dentro da array let valores e jogar la dentro pra somar, ela vai toda desorganizada e pois isso usamos o spread, ele vai espalhar os números la dentro, cada um dentro do local certo.

```
const soma=(v1,v2,v3)=>{  
  return v1+v2+v3  
}  
let valores=[1, 5, 4]  
console.log(soma(...valores))
```

.

Neste novo exemplo nos começaremos criando divs dentro do nosso JavaScript

```
const objs = document.getElementsByTagName("div")
```

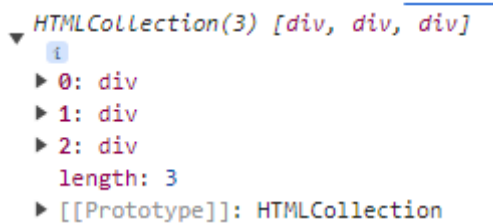
logo depois nos pegamos a tagname div e jogamos dentro de objs.

O document que tem em alguns codigos servem para o DOM

Se tentarmos

```
console.log(objs)
```

vai dar error pois isso não funciona no cscope ou editor de código pois isso seria pro lado do servidor mas devemos ver isso do lado do cliente, então podemos colocar lá no console do navegador que poderemos ver as divs e suas respectivas numerações



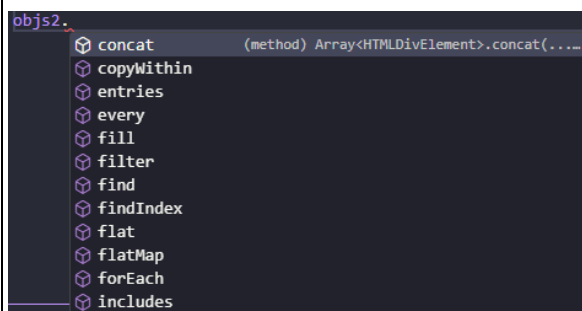
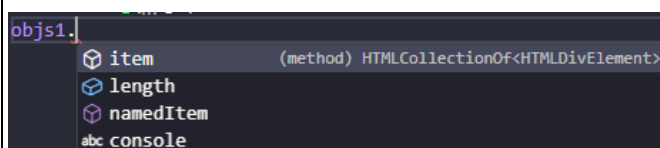
Faremos spread em nosso obj então da seguinte maneira

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

Como podemos ver o obj1 eo que fizemos anteriormente, agora o obj2 nos colocamos dentro de uma array e fizemos o spread, mas oque isso nos beneficiaria?

Isso dá uma lista mais longa de opções pois o obj1 que é o que chamamos de HTML collection que é do lado do cliente que é do html (lado do navegador), nos poderemos usar opções só do html, mas agora se fizermos ele como um array e colocarmos um spread nos poderemos utilizar ele também como uma array e com as funcionalidades de um array.

Podemos comparar isso logo abaixo, o tanto de opções que conseguimos utilizando o spread e fazendo ele como uma array.



Agora vamos supor que eu queira percorrer pelos elementos do obj1 e retornar o conteúdo usando forEach

```
const obj1 = document.getElementsByTagName("div")
obj1.forEach(element => {
  console.log(element)
});
```

```
► Uncaught TypeError: aula12.js:28
obj1.forEach is not a function
at aula12.js:28:7
```

Vai dar erro sem o spread

.

Mas agora vamos tentar fazer a mesma coisa com o obj2

```
const obj2 = [...document.getElementsByTagName("div")]
obj2.forEach(element => {
  console.log(element)
});
```

```
aula12.js:29
<div>Bruno</div>
aula12.js:29
<div>CFB Cursos</div>
aula12.js:29
<div>Novo curso de JavaScript</div>
```

Agora deu certo nos retornou(mostrou) os elementos html.

.

Agora nos podemos fazer uma melhor manipulação até do HTML

Podemos até mesmo mudar o texto das 3 divs que eu peguei e coloquei no obj2

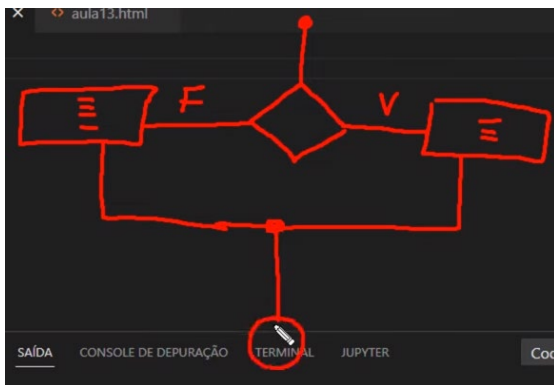
```
obj2.forEach(element => {
  element.innerHTML="curso"
});
```

```
curso
curso
curso
```

.

Comando condicional IF e IF ELSE em Javascript - Curso de Javascript Moderno - Aula 13

A condicional ela vai englobar uma expressão logica retornando true ou false, mas podemos escolher, então por exemplo se for retornado true eu posso escolher que execute um certo bloco de códigos, mas caso seja false ele continua o código normal.



Temos o diagrama acima para utilizar como exemplo, aqui lá no início do diagrama nós temos o código normal, mas quando chega naquele trapézio podemos dizer que aparece o teste condicional, então se o teste lógico for verdadeiro ele vai executar o código de nossa escolha representando no diagrama como o V do lado direito, mas se for false ele executará o código da esquerda e após isso ele termina de executar o código normal.

Agora faremos um exemplo de código mas da nossa maneira

```
Se (5>4){  
  Bloco de código 1  
}senão{  
  Bloco de código 2  
}
```

Isso é como se fosse o código, se 5 for maior que 4 ele executará o bloco de código 1 mas se não for ele executará o bloco de código 2, o resultado será o bloco de código 1 pois 5 é maior que 4.

Em código JavaScript ficará assim

```
if(5>4){  
  console.log(`Bloco de código 1`);  
}else{  
  console.log(`Bloco de código 2`)  
}
```

Podemos colocar a quantidade que quisermos de código entre as chaves.

Podemos tambem aumentar esse teste condicional colocando mais opções]

Veja o exemplo abaixo

```
let num= 10
Se (num <=3){
  Escreva("num e maior que 3")
}senaoSe (num >6){
  Escreva("num e maior que 6")
}senaoSe (num >7){
  Escreva("num e maior que 7")
}senão{
  Escreva("num e maior que todos os numeros")
}
```

```
let num= 10
if(num<=3){
  console.log('num e maior que 3')
}else if(num>6){
  console.log("num e maior que 6")
}else if(num>7 && num!=100){
  console.log("Num e maior que 7")
}else{
  console.log('nukm e maior que todos os numeros')
}
```

Aqui temos diversos teste condicionais, o teste que der true sera executado mas caso nenhum de certo o ultimo escrito else e executado.

PODEMOS COLOCAR TESTE CONDICIONAIS DENTRO DE TESTES CONDICIONAIS

Comando Switch Case em Javascript - Curso de Javascript Moderno - Aula 14

O Switch avalia uma expressão e de acordo com, o resultado dessa expressão, ele compara com os cases que ele ta programado para mostrar o resultado, e se por caso algum coincidir com os cases que ele tem o case e executado.



Esse e nosso diagrama

nesta parte e colocado a nossa expressao que queremos ai logo vamos para o case 1, ai se por acaso o primeiro case for igual a expressão que colocamos nesse prisma, o codigo dentro dele ser executado, mas caso não seja igual ele vai pro segundo case e dai por diante ate achar um case que tenha aver com a expressão colocada no topo, se não for encontrado nenhum sera executado aquele final do codigo escrito def. que significa (default) ele serve para caso não tenha nenhum case que atende os requisitos la da expressão esse default e executado, e depois continua o resto do codigo pra baixo e esse e nosso switch case.

Temos um exemplo de codigo abaixo

```
let colocacao = 1
switch(colocacao){
  case 1:
    console.log("Primeiro Lugar")
    break
  case 2:
    console.log("Segundo Lugar")
    break
  case 3:
    console.log("Terceiro Lugar")
    break
  default:
    console.log("Não subiu ao pódio")
    break
}
```

Aqui nos temos uma variável chamada colocação com o lugar que o jogador conseguiu ficar queo eo primeiro, podemos ver tambem que logo abaixo temos cases, são 3 cases e cada um

com uma característica, o que esse código faz é o seguinte caso colocação seja igual a 1 ele mostra "Primeiro Lugar", mas se caso colocação for igual a 2 o resultado será "Segundo Lugar", mas caso a colocação for igual a 3 será mostrado "Terceiro Lugar", mas se por um acaso colocação não for igual a nenhum desses casos será executado o default mostrando assim "Não subiu ao pódio", devemos lembrar que esses breaks dentro de cada case e do default é essencial para o funcionamento do código e os requisitos dos cases não precisam ser somente números, como no exemplo abaixo:

```
case "Legal":  
    console.log("Você é um cara legal")  
    break
```

Podemos colocar quantos cases quisermos

Podemos fazer o seguinte também

```
case 3: case 4: case 5:  
    console.log("Terceiro Lugar")  
    break
```

Vários cases para um só bloco de código.

.

Loop FOR em Javascript - Curso de Javascript Moderno - Aula 15

Uma estrutura em loop é uma estrutura que vai repetir um bloco de comandos ou um comando só, e sempre que precisarmos repetir a execução de um conjunto de instruções nos utilizamos um loop, cada repetição que o loop fizer nos estamos fazendo uma interação.

Os loops nós podemos classificar entre definidos e indefinidos, loops que vamos usar quando sabemos a quantidade de vezes que precisamos repetir ou os indefinidos quando não sabemos quantas vezes ao certo que vamos repetir.

Loops Definidos: For.

Loops indefinidos: While, Do While.

A sintaxe do for é a seguinte

```
For(inicialização; condição; contador){  
}
```

O que estiver dentro das chaves será repetido.

Temos o seguinte exemplo:

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

Aqui nós inicializamos criando uma variável e colocamos 0 dentro dela, depois criamos uma condição para a petição que fará que o loop continue enquanto i for menor que 10, e colocamos i++, onde sempre que a estrutura de loop repetir será incrementado 1 à variável i.

E o código que será executado dentro desse loop é o console.log(i), isso fará com que mostre o valor da variável a cada repetição, então resumindo enquanto i for menor que 10 a variável i será incrementada e mostrará o novo valor da variável i no console, uma outra explicação é que enquanto a condição for verdadeira(true) ele continuará o loop, então o resultado do código será uma contagem de 0 até 9.

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

.]

Loops FOR IN e FOR OF em Javascript

- Curso de Javascript Moderno - Aula 16

Os números em javascript sempre são contando começando do zero, então em qualquer contagem não será 1, 2, 3, 4 mas sim 0, 1, 2, 3, 4)

Primeiro devemos entender o `.length`.

`.length`

Faz com que não mostre o valor mas sim a quantidade, então se temos a seguinte lista array

```
let num=[10,20,30,40,50];
console.log(num.length)
```

temos 5 elementos dentro dela, então com o `num.length` invés de mostrar os números dentro da array, será mostrado a quantidade de números que temos dentro dessa array, e no exemplo acima nós temos 5 números então o valor retornado será 5.

nosso exemplo temos o seguinte código:

```
let num=[10,20,30,40,50];
for(let i=0; i<num.length; i++){
  console.log(i)
}
```

Esse código é o seguinte primeiro temos uma array, com alguns números [10, 20, 30, 40, 50], e logo embaixo temos uma estrutura de repetição FOR, onde na repetição colocamos uma inicialização que `i` recebe 0, aí em nossa condição enquanto `i` for menor que `num.length` o loop será repetido, e no contador nós colocamos um incremento que sempre quando tiver essa repetição haverá incremento no `i`, e será mostrado o valor de `i` no `console.log`.

Então o código será executado 5 vezes pois o valor de `num.length` é 5 porque tem 5 números dentro dessa array, mas se quisermos mostrar o valor dentro da array um por um temos uma técnica com o loop for

```
let num=[10,20,30,40,50];
for(let i=0; i<num.length; i++){
  console.log(num[i])
}
```

Já notou alguma diferença?

Sim dentro do `console.log` nós agora colocamos `num[i]`, você se lembra que para acessarmos um número de uma array precisamos de pegar o nome dessa array e colocar entre colchetes o índice da posição do elemento que estamos procurando?

Então olhando o array acima para acessarmos o número 30 nós devemos utilizar

```
console.log(num[2])
```

que é a posição dele no índice.

Entao agora e muito simples, como sempre que repetimos o loop e incrementado 1 a variavel i e a variável i esta dentro de num(i), entao cada hora sera mostrado uma posição dentro do índice de num, fazendo assim que seja mostrado todos números dentro da array num.

Resultado

```
Info: Start process (14:27:59)
10
20
30
40
50
Info: End process (14:27:59)
```

For In

Este código faz a mesma coisa do anterior

```
let num=[10,20,30,40,50];
for(n in num){
  console.log(num[n])
}
```

Esse for in faz o seguinte, ele primeiro pega e ve qual coleção que queremos percorrer, neste caso a coleção ea num que e nosso array, ai ele percorre toda essa coleção e coloca os índices dentro de n, que e como se fosse uma variável dentro desse loop colocando os índices que tem dessa coleção, entao sempre que esse loop percorre a array e encontra um elemento la dentro ele repete denovo, ate acabar os elementos, ai no final dentro de n temos a quantidade de elementos que existem dentro dessa array, e enquanto tudo isso acontece, a array em conjunto com seu índice esta sendo mostrado dentro do console.log mostrando assim todos os números.

Entao se eu quiser pegar as posições da array dentro de n nos usamos o for in.

Podemos fazer isso tambem no html

For of

```
for(n of num){
  console.log(n)
}
```

O for of e bem parecido com o for in, ele percorre toda a coleção e já conta automaticamente a quantidade de elementos que tem dentro da coleção e de acordo com quantidade de elementos será a mesma quantidade que ele será repetido, entao ele vai percorrendo, so que invés dele percorrer e colocar dentro de nos índices da coleção, ele já coloca os elementos direito, entao ele não vai colocar os índices, mas sim o conteúdo do índice até acabar todos.

Entao se eu quiser pegar o conteudo dos índices da array e colocar dentro de n nos usamos for of

For of e For in no navegador (DOM)

```
const objs =document.getElementsByTagName("div");
for(o of objs){
  console.log(o.innerHTML="curso")
}
```

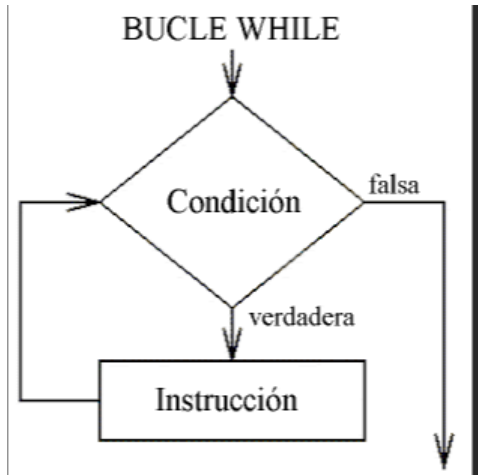
Podemos pegar as divs por exemplo de uma pagina html e colocar a palavra “curso” de acordo com a quantidade de divs.

.

Loop While em Javascript - Curso de Javascript Moderno - Aula 17

O loop while é um loop indefinido (Quando não temos certeza da quantidade de vezes que queremos executar esse loop).

Diagrama (While)



Enquanto a condição for verdadeira o loop continuara, mas apartir do momento que a condição se tornar falsa, o loop vai parar e sair para continuar outros códigos.

.

While = enquanto

```
let n=0
while(n<10){
  console.log(n)
}
```

Neste exemplo acima nos temos um loop infinito, primeiro nos criamos uma variável chamada n com o valor zero, depois começamos a estrutura de repetição while, e enquanto n for menor que 10, sera mostrado no console a variável n, mas sera infinito pois n sempre sera menor que 10 porque a variável n neste caso não muda, a variável n sempre sera 0 e sempre sera menor que 10.

Mas agora se incrementarmos algum valor a variável n ai nos podemos deixala finita invés de infinita.

```
while(n<10){
  n++
  console.log(n)
}
```

Agora neste novo exemplo nos estamos incrementando 1 a variável n, fazendo assim uma contagem no nosso console de 1 ate 10.

Mas so lembrando que esses códigos são exemplos, podemos usar a estrutura de repetição para outra diversas maneiras, podemos mudar a condição e etc.

.

Por exemplo podemos criar um programa para calcular o fatorial de 5

```
let n=5
let fat = 1
while(n>=1){
  console.log(fat)
  fat*=n
  n--
}
```

WHILE e DO WHILE em Javascript, entenda a diferença - Curso de Javascript Moderno - Aula 18

Ambos vão executar quando uma expressão for verdadeira, ambos são loops que nós vamos usar em situações indefinidas, que é quando nós não sabemos quantas vezes que vamos repetir o loop.

Essa seria a estrutura em português do do-while

```
Faça{
    Codigo aqui
}while(condição)
```

Então diferente do while comum, o do-while é executado ao menos uma vez, e se vai fazer um loop ou não depende se a expressão for verdadeira ou falsa, mas agora o while se a condição for falsa não será executada nenhuma vez e não será executado.

Então o Do-while primeiro executa o código e pode repetir se for verdadeiro, mas se for falsa não repete, e o while comum se for verdadeiro ele fará o loop normalmente mas se for falsa não é executado nenhuma vez, e simplesmente ignorada.

```
let n=1
do{
  console.log("teste")
}while(n=10)
```

Neste exemplo do do-while o console.log primeiro é executado e depois vai para a condição para ver se o código vai ser executado mais uma vez, mas não vai porque a variável n não é igual a 10 então segue o código mas já mostrando a palavra teste uma vez pois já foi executado.

Entenda as declarações BREAK e CONTINUE em Javascript - Curso de Javascript Moderno - Aula 19

break

O break é uma interrupção na execução, então encontrou o comando break, ele para o comando loop e continua a execução normal do código, ele para totalmente o loop.

```
let n=0;
let max=1000;
while(n<max){
  console.log("CFB Cursos - " + n);
  if(n>10){
    break;
  }
  n++
}

console.log("Fim do programa")
```

por exemplo no código acima nós temos a estrutura de repetição que enquanto n for menor que max ele continua sendo executado e max tem o valor 1000, então o comando seria executado 1000 vezes, mas temos uma estrutura condicional com if onde, quando n for maior que 10, ele tem um break, e quando cai no break o loop é parado de vez e termina a execução desse loop, fazendo assim que a contagem seja somente até 11.

.

Continue

O continue quando ele é executado dentro de um loop ele para somente aquela interação e pula para a próxima interação ele só não vai executar a interação atual e vai pular para a próxima, mas continua dentro do loop.

.

Aprendendo sobre FUNÇÕES em Javascript [#P1](#) - Curso de Javascript Moderno - Aula 20

Uma função é um bloco de comando que podemos executar em um momento que precisarmos ou escolhermos, vamos usar que estamos executando todo o código normalmente, mas aí criamos uma função no meio desse código, ele não será executado pois ele só é executado quando é chamado, então ele vai pular a função e vai continuar o código, mas quando a função for chamada aí sim ela será executada, temos diversas maneiras para executá-la, como um click, mover mouse, load e etc...

```
function nomeDaFunction(){  
}
```

Essa é uma estrutura de uma function onde escrevemos function para inicializar, e depois escolhemos um nome pra function e depois do nome sempre devemos colocar parênteses, esses parênteses são obrigatórios ter, mas não quer dizer que precisamos preencher ou usar mas os parentes precisam ficar lá mesmo que vazios, esses parênteses recebem um parâmetro, aí depois abrimos e fechamos chaves e dentro das chaves que colocamos nossos códigos, então se tiver rodando um código normal aí o código chegar em uma function, não vai acontecer nada ele simplesmente vai pular a function, mas quando chamamos ela da seguinte maneira

```
nomeDaFunction()
```

Aí quando o código chegar nessa parte, a function lá em cima vai ser executada, aí todo o código dentro da function vai ser executada.

Olhe o exemplo abaixo

```
function soma2_10(){  
  let n1=2  
  let n2=10  
  let soma= n1+n2  
  console.log(soma)  
}  
for(let i=0; i<10;i++){  
  soma2_10()  
}
```

Pode parecer um pouco complexo mas na verdade é bem simples, temos uma function e uma estrutura de repetição for que já aprendemos, se não tiver entendendo volte para as anotações de estrutura de repetição for, aí na estrutura de repetição enquanto a variável i for menor que 10 o loop será repetido, e será chamado a function

```
soma2_10()
```

então como está chamando a function, enquanto i for menor que 10 a function soma2_10 será chamada

FUNÇÕES com retorno em Javascript [#P2](#) - Curso de Javascript Moderno - Aula 21

Podemos usar o return da seguinte maneira, olha que interessante

O return não faz nada mais além de retornar algo, uma informação então veremos um exemplo abaixo

```
function canal(){  
  return "CFB Cursos"  
}
```

Nessa function canal(), nós temos um return e esse return vai retornar a string "CFB Cursos", nós poderíamos colocar dentro do console.log para poder aparecer no terminal, mas nós faremos dessa vez fazendo um retorno, esse código vai retornar a string "CFB Cursos", então quando pegarmos essa function e mostrar no console será mostrado o que foi retornando, ou melhor ele vai retornar o que está com return e será mostrado.

```
console.log(canal())
```

Aqui nós pegamos o console.log e dentro dele colocamos a function, o que será mostrado é óbvio, será mostrado o retorno que é "CFB Cursos", isso porque CFB Cursos recebeu um return

A função `canal()` retorna a string "CFB Cursos", e quando você chama `console.log(canal())`, o valor retornado pela função é exibido no console.

É uma forma útil de encapsular um pedaço de código que realiza uma operação específica e retorna um resultado, permitindo que você reutilize esse código em diferentes partes do seu programa.

FUNÇÕES parametrizadas em Javascript [#P3](#) - Curso de Javascript Moderno - Aula 22

Uma função parametrizada significa que queremos entrar com valores para dentro da função, nos entramos com os parâmetros dentro dos parênteses, e eles são executado dentro da function.

Exemplo:

```
function soma(p1){  
    console.log(p1)  
}
```

O valor dentro do parâmetro (p1) é colocado dentro da function para ser usado, e neste exemplo nós o utilizamos para ser mostrado no console.log(), mas antes precisamos dar um valor a esse parâmetro e mostraremos como.

Para colocar um valor no parâmetro nós podemos chamando a própria function

```
function soma(p1){  
    console.log(p1)  
}  
soma("Cfb Cursos - JavaScript")
```

neste exemplo criamos a function com o parâmetro p1 e dentro da function o console.log vai imprimir o conteúdo de p1, aí embaixo da function temos uma chamada, que chama a function para ser executada, mas assim que é chamada entre os parênteses temos o valor que será atribuído ao nosso parâmetro p1, então ao mesmo tempo que chamou a function essa última linha de código também atribuiu um valor ao parâmetro p1.

E toda vez que chamarmos a function atribuindo um novo valor ao parâmetro, ele será executado e mostrado todas as vezes.

```
function soma(p1){  
    console.log(p1)  
}  
soma("Cfb Cursos - JavaScript")  
soma("E um ótimo curso")  
soma("John CHK")
```

Resultado

```
Info: Start process (23:58:29)  
Cfb Cursos - JavaScript  
E um ótimo curso  
John CHK  
Info: End process (23:58:30)
```

Podemos colocar 2 parametros tambem

```
22.js > ...  
function soma(n1,n2){  
  console.log(n1+n2)  
}
```

Neste exemplo nos criamos uma function chamada soma com 2 parametros: n1 e n2 e dentro dessa function temos uma operação de soma, entre a soma de n1 e n2, mas ate o momento não temos valores para nossos parâmetros.

```
22.js > ...  
function soma(n1,n2){  
  console.log(n1+n2)  
}  
  
soma(10,5)
```

Entao agora nesta nova imagem nos chamamos a function e colocamos 2 valores para os parâmetros, entao colocamos o numero 10 para o n1 e o numero 5 para o n2.

O resultado da soma sera 15

.

E podemos passar quantos parâmetros quisermos

Mas e se caso não passarmos nenhum valor para um parâmetro?

Caso não passemos nenhum valor ao parâmetro mas mesmo assim utilizamos ele, sera mostrado Nan pois não tem valor, mas para resolver isso podemos passar valores padrões para eles, entao caso um parâmetro não tenha sido atribuído nenhum valor podemos colocar valores padrões

Exemplo

```
function soma(n1=0, n2=0){  
  console.log(n1+n2)  
}  
  
soma(2)
```

Neste exemplo acima passamos o valor apenas de um parâmetro, mas colocamos valores padrões a eles, entao como colocamos o valor 10 a n1 e o n2 já esta com 0 por padrão, o resultado da soma sera 2.

.

Podemos utilizar return tambem

```
function soma(n1=0, n2=0){  
  return n1+n2  
}  
  
console.log(soma(2))
```

primeiro vai ser chamado a function levando o numero 2 ao parâmetro n2 depois a soma sera feita e retornara o resultado, ai como chamamos o console.log na chamada da function, sera mostrado o valor em nosso console, e o resultado continua sendo 2.

.

Outra situação

```
function soma(n1=0, n2=0){  
    return n1+n2  
}  
let resultado_soma =soma(5,5)  
console.log(resultado_soma)
```

Essa e a mesma coisa mas a function será chamada vai executar a function fazendo o calculo, ai o return vai retornar o valor e jogara dentro da variável resultado_soma, e depois vai ser imprimido no console.log().

.

Outra situação

```
2  
3  function add(v){  
4      return valor+v  
5  }  
6  
7  let valor=0  
8  console.log(valor)  
9  
10 valor=add(10)  
11 console.log(valor)  
12  
13 valor=add(5)  
14 console.log(valor)  
15
```

PROBLEMAS SAÍDA CONSOLE DE DEPU

[Warning] node "d:\novo Javascript

0
10
15

.

Parâmetros REST em funções Javascript [#P4](#) - Curso de Javascript Moderno - Aula 23

Quando criamos as function com o no exemplo abaixo]

```
function soma(n1=0,n2=0){  
    return(n1+n2);  
}  
console.log(soma(10,5))
```

Nos temos dois parâmetros, mas somente dois parâmetros, e cso queira colocar mais eu tenho que especificar os parâmetros dentro da function, entao o REST veio para resolver isso, poderemos ter quantos parâmetros nos quisermos sem precisar especificar cada um.

```
function soma(...valores){  
    return valores.length;  
}  
console.log(soma(10,5));
```

Aqui neste exemplo acima nos criamos um rest chamado valores, o rest e criado com 3 pontos na frente do nome, logo ali dentro da function criei um return que vai retornar o legth da REST chamada valores, entao ele vai retornar a quantidade de parâmetros que contem a REST valores, pois uma REST armazena apenas parâmetros, mas la em baixo no console.log chamamos a function chamada soma e colocamos dois valores para os parâmetros, e como colocamos dois valores para o REST logo o REST criou dois parâmetros para armazenar esses dados, e entao o resultado daquela function sera 2, pois oque foi retornado no console.log e o length de valores que e 2.

```
function soma(...valores){  
    return valores.length;  
}  
console.log(soma(10,5,6,7,2));
```

Neste novo exemplo acima temos 5 valores para os parâmetros, entao o REST criara 5 parametros e armazenara cada um desses valores, e o resultado que sera retornado vai ser 5 pois são 5 valores que teremos.

Essa function abaixo calcula todos os valores que tem no parâmetro valores

```
function soma(...valores){  
  let tam = valores.length;  
  let res=0;  
  for(let i=0; i<tam;i++){  
    res+=valores[i]  
  }  
}  
console.log(soma(10,5,6,7,2));
```

Primeiro criei uma variável chamada tam que armazena o numero de valores que tem a REST valores, entao depois cria uma variável chamada res que tem o valor padrão de 0 mas dentro do looping for ele recebe cada valor dos parametros e soma tudo, e no looping for enquanto i ser menor que tam, que é a quantidade de parametros dentro de valores o for continua rodando ate tam seja do tamanho de i e logicamente tenha passado por todos os parametros de valores.

.

Podemos faze-lo usando for of tambem que já aprendemos a usar antes

```
function soma(...valores){  
  let res = 0  
  for (let n of valores) {  
    res+=n  
  }  
  return res  
}  
console.log(soma(10,20,30))
```

Agora utilizando tambem o for of podemos somar os valores de valores.

.

Funções Anônimas em Javascript [#P1](#) - Curso de Javascript Moderno - Aula 24

São funções que não tem nomes associados ao seu corpo ou seu conteúdo, são funções que são chamadas em tempo de execução, ela só é criada no momento da execução, ela não fica pronta na memória antes de ser chamada.

Uma função comum ela é criada e fica guardada esperando somente que seja chamada em alguma parte do código.

Exemplo de função anônima

```
const f=function(v1,v2){  
    return v1+v2;  
}  
console.log(f(10,5))
```

A diferença neste exemplo é que na função anônima não precisamos usar um nome para chamar a função anônima.

Construtor

Toda vez que falarmos sobre construtor devemos nos lembrar do operador new, que pode mudar a construção da nossa aplicação, ele serve para operações mais simples.

```
const f=new Function("v1","v2","return v1+v2")  
console.log(f(10,5))
```

Dois detalhes, primeiro colocamos o new antes do function e o function deve iniciar com F maiúsculo, neste exemplo acima estamos usando uma function construtor dentro de uma function anônima.

Precisamos passar os argumentos, e o corpo da função, neste exemplo os argumentos são o v1, v2, v3 e o corpo da função será o return v1+v2.

O que manda é sempre o último parâmetro da função construtor, pois ele será o corpo da função, resumindo será o que a função vai fazer com esses parâmetros.

Arrow Function em Javascript.

Aprenda o que é e como usar! - Curso de Javascript Moderno - Aula 25

Funções lambdas (arrow function)

E um tipo de declaração de função na anonima,

```
const soma=function(v1, v2){return v1+v2} //Função comum
```

```
const soma=(v1,v2)=>{return v1+v2} // arrow function
```

Neste exemplo acima temos um exemplo de function comum e uma arrow function, os dois códigos trazem o mesmo resultado so que com técnicas diferentes, podem ver que tiramos o function que estava escrito, e também apontamos com uma seta => para onde a function seria executada.

Se for uma função que vai entrar so um parâmetro não precisamos colocar parênteses, ficando assim

```
const nome = n=>{return n}
```

Na verdade no padrão array function nem precisaríamos do return, e se não tiver return na array function não precisamos nem das chaves

```
const add=n=>n+10
```

ficando desta maneira, mas isso continua sendo uma function.

Mas é claro que não usaremos chaves nem parênteses somente se for um código bem simples, que precise de so uma linha, senão teremos de usar toda a sintaxe dele.

```
const add=n=>n+10  
console.log(add(10))
```

Podemos ver acima, primeiro nós criamos ali a array function sem usar parênteses nem chaves, a array function faz uma soma de n+10, mas aí quando colocamos o parâmetro ali dentro do console.log, foi adiciona 10 a variável n ficando assim o resultado se tornou 20.

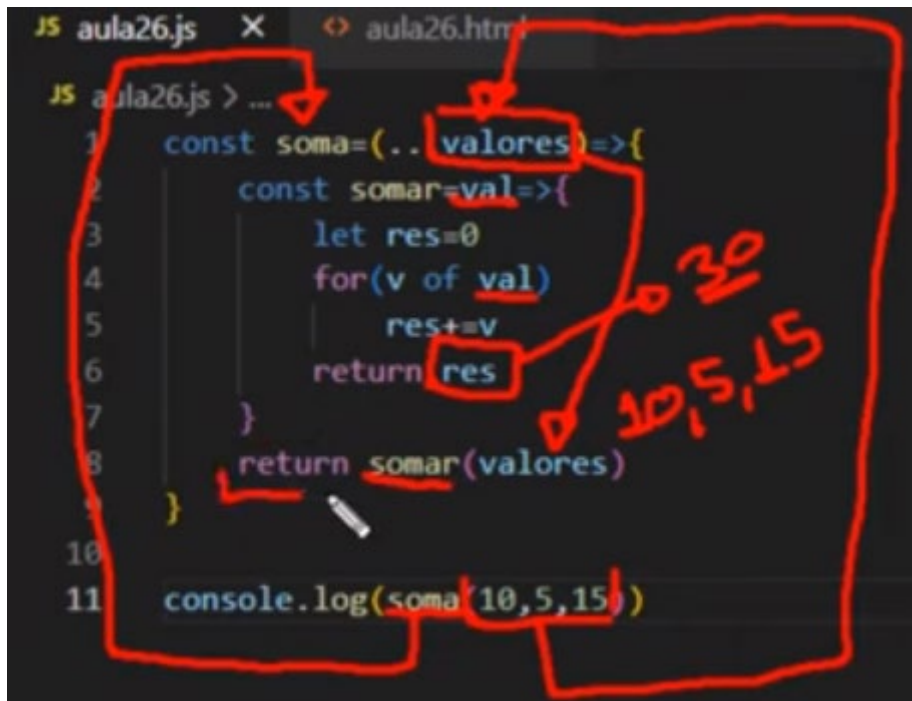
Sintaxe completa do array function

```
let variavel = (parametros)=>{  
  /* CODIGO */  
}
```

Funções dentro de funções em Javascript. Funções aninhadas - Curso de Javascript Moderno - Aula 26

Temos aqui uma função dentro de outra função

```
const soma=(...valores)=>{  
  const somar=val=>{  
    let res=0  
    for(v of val){  
      res+=v  
      return res  
    }  
  }  
  return somar(valores)  
}  
console.log(soma(10,5,15))
```



Funções Geradoras em Javascript.

Você precisa aprender! - Curso de Javascript Moderno - Aula 27

Uma função geradora tem seu retorno adiado até que nos precisemos desse retorno, ou melhor essa função é pausada mantendo os valores dentro dela até que precisemos dela novamente, então a execução nunca é parada mas sim pausada, então sempre que executarmos essa função e quisermos voltar mais tarde e executá-la novamente o valor que estava lá dentro antes ainda estará lá, e ela é o pilar da programação assíncrona, quando uma função normal é chamada quem fica com o controle dessa função é a própria função, ela tem o seu controle até o final que ela retorna alguma coisa, mas as funções geradoras a própria função pode ter esse controle quando quiser retornar alguma coisa, resumindo não precisa de todo o processamento para retornar algo no final, então podemos ir retornando coisas ao longo da função, e toda vez que chamarmos a função ela pode retornar algo diferente.

Para criar uma função geradora precisamos usar a palavra `function*` com um asterisco no código, `yield` retorna o conteúdo no código.

```
function* cores(){
  yield 'Vermelho'
  yield 'verde'
  yield 'Azul'
}
const itc = cores()
console.log(itc.next().value)
```

Então no código acima, temos uma função geradora que deverá retornar Vermelho, verde e azul, lá em baixo em

```
const itc = cores()
console.log(itc)
```

já fizemos uma primeira chamada para a função que no `console.log` aparecerá assim

```
Object [Generator] {}
```

Nos mostra que o conteúdo da variável `itc` que é a função `cores()` é um `Object [Generator] {}`, mas agora para retornarmos os valores dessa função precisaremos de um outro chamado

```
console.log(itc.next().value) // Vermelho
```

Esse chamado com o `.next()` que é um método da função geradora, fará uma execução em nosso código retornando vermelho em nosso console. Agora se chamarmos a função mais duas vezes retornará verde e azul

```
console.log(itc.next().value) // Vermelho
console.log(itc.next().value) // Verde
console.log(itc.next().value) // Azul
```

mas e se agora chamarmos a função novamente o valor sera undefined pois não tem mais valores para retornar..

Novo exemplo

```
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('Bruno').value)
console.log(itp.next('Natação').value)
```

neste exemplo nos criamos uma função de retorno chamada perguntas(), e dentro dela criamos duas variáveis que receberão valores que serao inseridos nos chamados, na primeira execução, resumindo o resultado sera

```
Qual seu nome?
Qual seu esporte favorito?
Seu nome é Bruno, seu esporte favorito é Natação
```

.

Outro exemplo

```
function* contador(){
  let i=0
  while(true){
    yield ++i
  }
}

const itc = contador()
console.log(itc.next().value) // 1
console.log(itc.next().value) // 2
console.log(itc.next().value) // 3
console.log(itc.next().value) // 4
```

Neste exemplo, aquela estrutura de repetição se não fosse uma função geradora seria infinita, mas nesse caso eele retornara mais um cada vez que chamarmos essa função pois essa função mantem o valor anterior sempre, entao antes era 1 e depois a execução foi pausada, a próxima vez que chamarmos manetera o 1 e colocara mais 1, entao a função foi executada 4 vezes ai.

```
function* contador(){
  let i=0
  while(true){
    yield i++
    if(i>5){
      break
    }
  }
}
const itc = contador()
for(let c of itc){
  console.log(c) // 1 ate 5
}
```

Aqui terá uma contagem ate 5, pois na verdade isso e uma execução in finita mas como dentro do while determinamos que apartir do momento que a variável i valer 5 a execução sera terminada usando o break.

.