**Tratamento de dados - Curso JavaScript #06**

**Tipos primitivos e seus valores relacionados**

- Number (infinity, NaN)

- String

- Boolean

- Null

- Undefined

- Objetct (array)

- Function

**Tipos de dados**

**+** tem valor de CONCATENAÇÃO e ADIÇÃO.

number + number = adição;

string + string = concatenação.

**Para converter string para número inteiro**: Number.parseInt(n)

**Para número real**: Number.parseFloat(n)

**Para o JS decidir se é inteiro ou real**: Number(n)

**Exemplo:** antes de jogar dentro de n1 e n2, estou convertendo para inteiro. A variável N1 vai receber o que vier do prompt da janela, convertido para o número inteiro. Vejamos:

<script>

var n1 = Number.parseInt(window.prompt("Digite um número: "))

var n2 = Number.parseInt(window.prompt("Digite outro número: "))

var s = n1 + n2

window.alert("A soma dos valores é: "+ s)

</script>

**Para converter número para string**

String(n): assim como utilizei Number(n), assim gera uma string

n.toString(n): assim eu consigo girar e jogar n para string

**Formatando Strings**

Var s = ‘JavaScript’

‘Estou aprendendo s’ // não faz interpolação

‘Estou aprendendo’ + s // usa concatenação

`Estou aprendendo ${s}` //template string, onde **$** chame-se ***placeholder***

**Exemplo:**

Nome = ‘gustavo’

idade=41

nota = 5.5

Com concatenação, o resultado é:

'O aluno ' + nome + ' com ' + idade + ' anos tirou a nota ' + nota

Usando template string, fica:

`O aluno ${nome} com ${idade} anos tirou a nota ${nota}`

**Mais funcionalidades com string**

Existem outras coisas que posso fazer com string, por exemplo:

s.length // quantos caracteres a string tem

s.toUpperCase() //tudo para ‘MAIÚSCULA’

s.toLowerCase() //tudo para ‘minúsculas’

**Exemplo:**

<body>

<script>

var nome = window.prompt('Qual é o seu nome? ')

document.write(**`**Olá, ${nome}! Seu nome tem ${nome.length} letras.<br/>**`**)

document.write(**`**Seu nome em maiúsculas é ${nome.toUpperCase()}<br/>**`**)

document.write(**`**Seu nome em minúsculo é ${nome.toLowerCase()}**`**)

</script>

</body>

**Formatando números**

> var n1 = 1545.5

> n1.toFixed(2)

'1545.50'

> n1.toFixed(2).replace('.',',')

'1545,50'

> n1.toLocaleString('pt-BR', {style: 'currency', currency: 'BRL'})

'R$ 1,545.50'

**Operadores (Parte1) - Curso JavaScript #07**

**Operadores**:

São cinco: Aritméticos; Atribuição; Relacionais; Lógicos; Ternário.

1. **Aritméticos**

|  |  |
| --- | --- |
| Esses operadores são binários, ou seja, precisam de dois operandos:  C:\Users\Angélica\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\7DA32BE8.tmp | Ordem de Precedência:  *C:\Users\Angélica\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\758C236.tmp*  Quanto à ordem de precedência dos operadores aritméticos, caso exista uma expressão com “**\***”, “**/**” e “**%**”, a ordem a ser executada é da esquerda para direita. |

1. **Atribuição**

|  |  |
| --- | --- |
| Simples:  C:\Users\Angélica\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\154D03B4.tmp | Auto-atribuição:  C:\Users\Angélica\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\B8A587E2.tmp  Tudo isso que aconteceu aqui, fez a variável partir de 3 e terminar com 16. |
| Simplificação:  Posso simplificar as atribuições se N receber ele mesmo.  C:\Users\Angélica\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\69A55240.tmp | Incremento:  C:\Users\Angélica\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\E5CBB24E.tmp  *var n = 10*  *n++: 10 //incrementa depois*  *n: 11*  *var n = 10*  *++n: 11 //pré-incremento, já soma antes* |

**Operadores (Parte2) - Curso JavaScript #08**

**Operadores**:

Veremos os operadores Relacionais; Lógicos e Ternário.

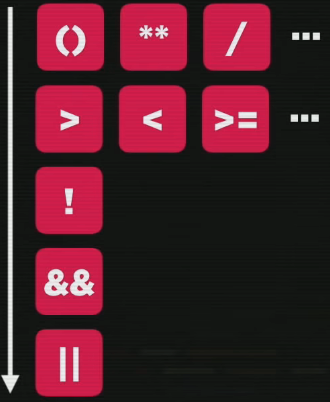
1. **Relacionais**

|  |  |
| --- | --- |
| Para toda expressão que tenha um valor relacional ligado a ela, o resultado vai ser sempre booleano.  C:\Users\Angélica\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\EAB6238C.tmp | Operadores de identidade:  “**==**” quer dizer comparando o valor, sem testar o tipo. “**=**” é recebe.  O operador de identidade, conhecido como igualdade restrita é “**===**” e comparar valor e tipo, ou seja, se um valor é idêntico.  C:\Users\Angélica\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\C1BD2D7A.tmp |

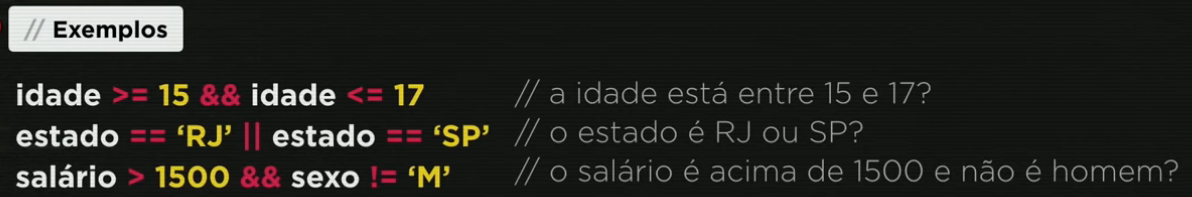
1. **Lógicos**

|  |  |
| --- | --- |
| C:\Users\Angélica\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\CB044398.tmp  ||: chama-se pipe | C:\Users\Angélica\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\FF74A566.tmp  C:\Users\Angélica\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\E7DB3E64.tmp  C:\Users\Angélica\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\7C218612.tmp |

Ordem de precedência:



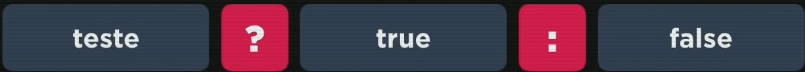
|  |
| --- |
| Quando eu tenho operadores lógicos, relacionais e aritméticos na mesma expressão, primeiro faço os aritméticos (%), depois os relacionais (== e >) e por fim os lógicos (&&).  5 **>** 8 **&&** 8 **%** 2 **==** 0  *8%2 dá 4, resto 0. Esse lado é verdade.*  *5 não é maior que 8.*  *Falso e verdadeiro, dá falso.*  Lembrando que os operadores aritméticos têm ordem de precedência também, os relacionais não (quem aparecer primeiro, vai ser feito primeiro) e os lógicos também têm ordem de precedência. |



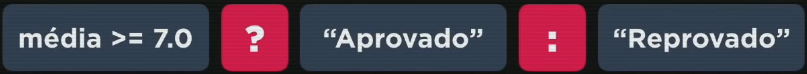
1. **Ternário**



O operador ternário é a interrogação e os dois pontos. Ele se chama ternário porque possui três blocos:



O primeiro é um teste lógico, que dá verdadeiro ou falso; o do meio é o que vai acontecer quando o teste lógico for verdadeiro; no fim, o que acontecerá quando o teste lógico for falso.



*Média é maior ou igual a 7? Se for, vai mostrar aprovado; se não, vai mostrar reprovado.*

**Introdução ao DOM - Curso JavaScript #09**

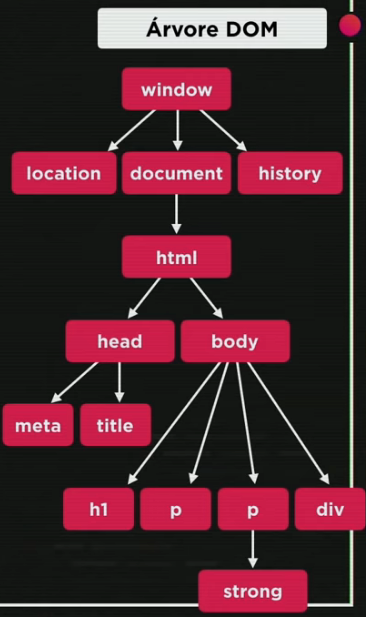
DOM é um acrônimo para **Document Object Model**, que é um modelo de objetos para documentos. É um conjunto de objetos dentro do navegador que vai dar acesso aos componentes internos do website.

O DOM não funciona dentro do NodeJS, por exemplo, ele está presente quando estou rodando JS dentro do navegador.

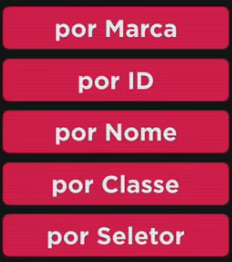
Uma coisa importante que devemos saber é fazer a Árvore DOM do site. Essa árvore começa da raiz, que se chama window. Então, tudo dentro do JS está dentro de um objeto chamado window. Então, a janela do navegador é um objeto DOM e se chama window.

Dentro do window, tenho vários outros objetos/elementos, como: Location (que diz qual é a localização do site: URL, qual é o objeto atual, qual é a página anterior), Document (documento atual) e History (que guarda de onde veio e pra onde vai).

Dentro do document, existe o objeto HTML. Dentro do HTML, tenho dois filhos: head e body. Ou seja, head e body são *child* e HTML é *parent*. Dentro de head temos várias tags. Dentro de body temos h1, dois parágrafos e uma div, no nosso documento atual. Dentro do segundo parágrafo ainda há um Strong.

****

Os elementos podem ser acessados dentro do JS utilizando o DOM por 5 métodos:



**Seleção por marca ou tag name**

**getElementsByTagName()**

Com esse método, consigo selecionar mais de um objeto do mesmo tipo/da mesma tag. Para selecionar especificamente, preciso incluir [] e dentro deles o index. Por exemplo, se tenho 3 parágrafos e quero selecionar o primeiro, uso [0].

<script>

*var* p1 = window.document.getElementsByTagName('p')[0]

</script>

**Seleção por ID**

**getElementById()**

Pode identificar um paragrafo ou uma div, por exemplo, e usar essa marcação. Usado principalmente quando o site é mais evoluído e não preciso selecionar um por um. Quando tem “s” no final preciso usar colchetes [ ].

**Seleção por nome**

**getElementsByName()**

Se eu tenho uma propriedade *name*, posso usar tal seleção. Esse tem “s” no final e para isso preciso usar colchetes [ ].

**Seleção por classe**

**getElementsByClassName()**

Posso usar como conjunto por classe.

Quando tenho mais de um objeto da mesma classe ou do mesmo nome, uso muito por marca ou por nome. Quando quero selecionar um objeto ou uma família deles, vou usar o ID.

Por exemplo, se eu tiver muitas tags divs, posso usar um ID.

<div *id*='mensagem'>clique em mim</div>

<script>

*var* d = window.document.getElementById('mensagem')

d.style.background = "green"

d.innerText = "Estou aguardando..."

</script>

**Seleção por Seletor**

**querySelector()**

**querySelectorAll()**

Forma nova, recomendável pela maioria dos manuais, faz a seleção através de elementos CSS. Utiliza-se querySelector, sendo o querySelectorAll para o plural. Navegadores mais antigos não vão ter suporte a ele.

<div *id*='mensagem'>clique em mim</div>

<script>

*var* d = window.document.querySelector('div#mensagem')

</spript>

div *class*='mensagem'>clique em mim</div>

<script>

*var* d = window.document.querySelector('div.mensagem')

<script>

ATENÇÃO:

Toda **classe** é representada por um “**.**”

Todo **id** é representado por uma “**#**”.

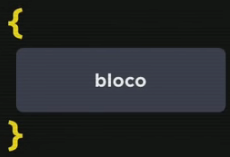
**Eventos DOM - Curso JavaScript #10**

Relembrando, DOM (Document Object Model) é um facilitador que existe dentro do navegador, dentro do HTML5, para dar acesso aos elementos diretamente para o JS. Sem o DOM, teríamos muita dificuldade para acessar os elementos da página.

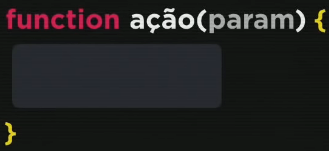
Evento DOM é qualquer evento que possa acontecer com os elementos. Por exemplo, tenho uma *div*. Um evento comum é o evento de mouse: posso mover o mouse até encostar/chegar dentro da *div* e quando isso acontecer, vai disparar o evento chamado ***mouseenter***. Ao continuar movendo o mouse por dentro da *div*, o evento ***mousemove*** vai ser disparado. Se eu pressionar o mouse, vai acontecer o disparo ***mousedown***. No momento que eu soltar o botão do mouse, chamodo evento ***mouseup***. O movimento de clicar vai disparar o evento ***mouseclick***. E já o ao mover o mouse para fora da *div*, vai disparar o evento ***mouseout***.

Para tratar um evento, precisamos entender o que é uma **função**. Uma função é um conjunto de códigos que vão ser executadas só quando o evento ocorrer. Então, por exemplo, ao programar um bloco de 10 linhas, esse bloco não vai ser executado automaticamente, só vai ocorrer, por exemplo, quando eu clicar dentro da *div*.

Para que eu dispare essas 10 linhas somente quando o evento ocorrer, o primeiro passo é coloca-las dentro de um bloco. Um bloco é delimitado entre os sinais de chaves.



Esse bloco tem que ser nomeado como uma *function*, então coloca-se a palavra *function* antes do boco. No JS, existe a função anônima, que é uma função que simplesmente não tem nome. Só que para que o método funcione, tenho que nomeá-lo. Geralmente, os nomes das funções são ações que podemos fazer, ações de eventos. Esses nomes são colocados depois de *function* e com parênteses. Opcionalmente, dentro dos parênteses posso passar algum(ns) parâmetro(s).



Os eventos podem ser configurados na parte do HTML ou no JS, o programador que escolhe a técnica que prefere.

**Exemplo:**

Quando clicar em cima da *div*, vai disparar o evento clicar através do método onclick=”clicar()”. No JS, adiciono a função clicar para configurar o que vai acontecer quando ela for chamada quando eu clicar.

<body>

<div *id*="area" *onclick*="clicar()">

Interaja...

</div>

</body>

<script>

*function* clicar() {

*var* a = window.document.getElementById('area');

a.innerText = "clicou!"

}

</script>

O nome do *getElementById* é igual ao ID. Sendo assim, o objeto ***a*** está ligado diretamente com a div/o elemento *id=área* (que está dentro do HTML) através de *window.document.getElementById('area');*. Vejamos mais interatividade com a div:

<body>

<div *id*="area" *onclick*="clicar()" *onmouseenter*="entrar()" *onmouseout*="sair()">

Interaja...

</div>

</body>

<script>

*function* clicar() {

*var* a = window.document.getElementById('area');

a.innerText = "clicou!";

a.style.background = "red";

}

*function* entrar(){

*var* a = window.document.getElementById('area');

a.innerText = "entrou!";

}

*function* sair(){

*var* a = window.document.getElementById('area');

a.innerText = "saiu!";

a.style.background = "blue";

}

</script>

Além de eu disparar diretamente pelo HTML, posso disparar os eventos utilizando ***listenners*** (ouvidores, que prestam atenção) pelo JS para o código HTML não ficar muito poluído. Assim, meu HTML fica limpo e meu JS que vai tratar de fazer essa ligação.

<body>

<div *id*="area">

Interaja...

</div>

</body>

<script>

*var* a = window.document.getElementById('area');

a.addEventListener("click", clicar);

a.addEventListener("mouseeenter", entrar);

a.addEventListener("mouseout", sair);

*function* clicar() {

a.innerText = "clicou!";

a.style.background = "orange";

}

*function* entrar(){

a.innerText = "entrou!";

}

*function* sair(){

a.innerText = "saiu!";

a.style.background = "purple";

}

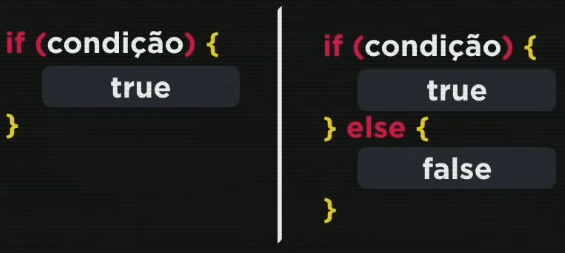
</script>

**Condições (Parte 1) - Curso JavaScript #11**

Condição é uma estrutura de controle muito importante para a programação. Podemos representar esse conceito como mostrado a seguir:



Existem vários tipos de condição. A seguir, veremos dois dos principais tipos:



Na primeira, condição simples, se a condição for falsa, nada vai acontecer em especial. Já na condição composta, caso seja verdadeira tem algo para fazer e caso seja falsa tem outra coisa a ser feita.

Condição simples:

*var* vel = 60.5

console.log(**`**a velocidade do seu carro é ${vel} km/h**`**)

if (vel > 60) { //condição simples

*//se a velocidade for maior que 60, abro o bloco e executo o comando*

console.log("Você ultrapassou a velocidade permitida. MULTADO!")

}

console.log("Dirija sempre usando cinto de segurança!")

Condicional composta:

*var* país = "EUA"

console.log(**`**Vivendo em ${país}**`**)

if (país == "Brasil") {

console.log("você é brasileiro")

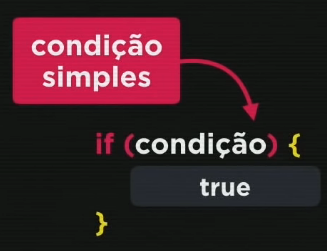
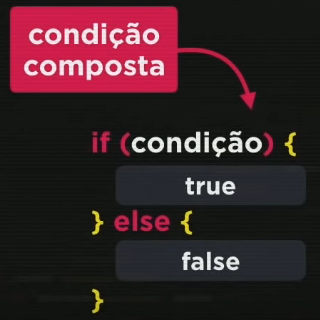
} else {

console.log("você é estrangeiro")

}

**Condições (Parte 2) - Curso JavaScript #12**

Temos dois tipos de condições: **Simples** e **Compostas**. Simples é *if* sem a cláusula *else*. Se a condição for satisfeita, vai executar um bloco. Agora, se eu precisar de blocos duplos, se uma determinada condição for verdadeira e se não faz outra, tenho condições compostas.

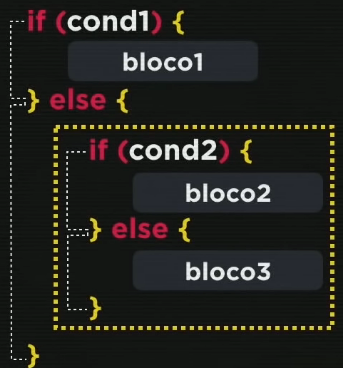
 

Existem situações em que várias formas de tratar um determinado dado. Para essas situações, usamos as **Condições Aninhadas**: pego uma condição composta e coloco outras condições dentro dela. Isso amplia muito as minhas possibilidades.

No exemplo se seguir, se a condição for verdadeira, executo o bloco ao lado esquerdo, se não, crio outra condição. E de novo, se for verdade eu faço uma condição, se não eu faço outra. Já

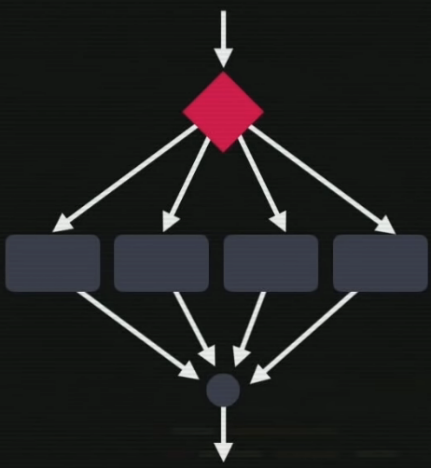


No exemplo abaixo, se a condição 1 for falsa, não vou fazer outro bloco, vou fazer outra condição. Se as condições 1 e 2 forem falsas, faço o bloco 3, mas nada me impede de no lugar do bloco 3 colocar outra condição, estaria aumentando o nível do meu aninhamento.



Além da estrutura de Condições Aninhadas, existe a estrutura de **Condições Múltiplas**. Muito útil no mundo da programação, ela serve para trabalhar com valores fixos e não serve muito para intervalos de valores (como bom dia, boa tarde, boa noite).

Ela tem a possibilidade não só do sim e não, ela tem a possibilidade de outros valores –– se for um valor, faz um bloco; se for outro, faz outro etc.

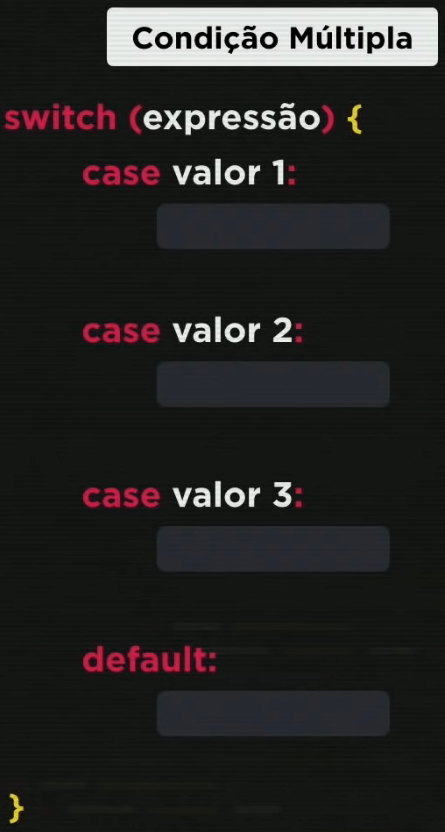


Então, eu tenho uma expressão e dessa expressão eu testo vários valores e depois eu volto para o fluxo normal do meu programa. Como dito, ela é muito útil para situações pontuais/específicas.

Essa estrutura é representada da seguinte maneira: temos um comando dentro do JavaScript que é o comando ***switch***.

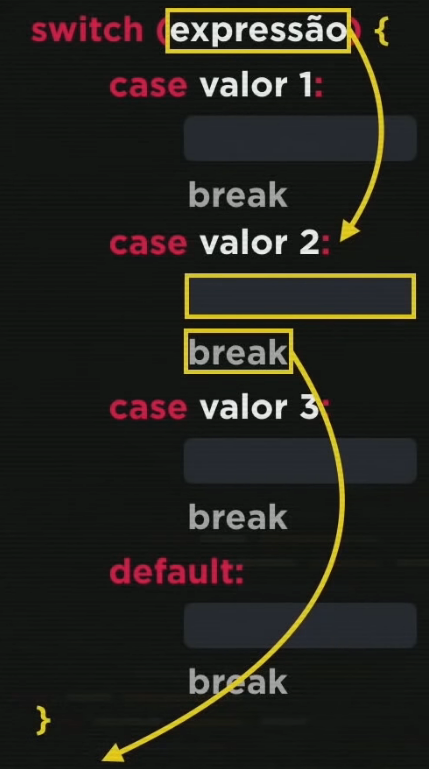
Esse *switch* tem uma expressão e um bloco relacionado com seus valores, que têm várias possibilidades e para cada um vou colocar um ***case***: se essa expressão for o primeiro valor, faz ele; se for o segundo valor, faz ele; etc. E para cada um dos cases, eu executo um bloco.

Tem uma cláusula no final (opcional) que se chama *default*, que é como se fosse o *else*: se nenhum dos valores forem satisfeitos, ele executa esse *default*.



Dentro da estrutura *switch*, tem um detalhe que precisa ser seguido: para o fim de cada bloco, eu **tenho que inserir obrigatoriamente um break**.

Como funciona o *break*: ele testa a expressão. Suponhamos que resulte no valor 2. A seta amarela desvia para o valor 2, executa os comandos do bloco e no final ele bate no *break*. Quando isso acontece, o fluxo vai ser desviado lá para baixo.



Se por acaso o *break* não existir, vai dar problema porque vai executar todos os comandos até achar um *break.*

O *switch* não funciona para intervalos (nesse caso, é mais valioso utilizar o *if*), só para situações pontuais e só funciona com números inteiros e caracteres. Então, *switch* é mais limitado que o *if*, mas ajuda em situações pontuais.

**Repetições (Parte 1) - Curso JavaScript #13**

As **estruturas de repetições** podem ser chamadas de **laços e iterações**. Na nossa vida, existem vários momentos em que existem repetições, como comer uma pizza.

Minha *function* é, tipo, uma vez por mês eu como uma pizza inteira. E para isso, tenho que fazer alguns passos, como comer a primeira fatia. Depois comer outra. E assim até a pizza acabar. A minha pizza tinha 4 pedações e, por isso, tenho 4 comandos “comer pizza”.

*function* comerPizza(){

comerFatia();

comerFatia();

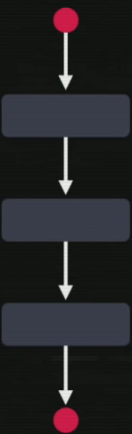
comerFatia();

comerFatia();

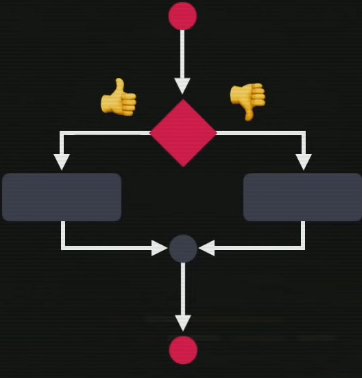
}

Então, minha atividade é partir do primeiro comando até o último comando pra cumprir minha missão. Na programação também é assim: **a gente parte de um ponto A até o ponto B, que é atingir o objetivo**. E para isso, existem várias **estruturas de controle**:

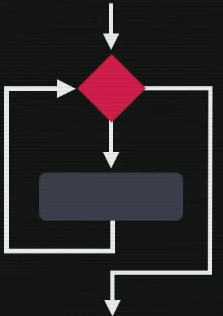
**Sequência**: Vou desde o ponto A até o ponto B executando tarefas sequencialmente até chegar ao objetivo, que é o aconteceu no caso de comer pizza.



**Condições (if / switch)**: Nem tudo funciona diretamente de A para B, às vezes acontecem desvios e, por isso, condições. Nas condições, tenho um teste lógico (representado por losango), que tem duas possiblidades: verdade ou falso. E de acordo com esse valor, são executadas tarefas específicas. Por exemplo, tenho pizza de 8 pedaços e não gosto de marguerita. Se a pizza for de marguerita, eu como, se não for, eu não como. E assim consigo chegar de um ponto até o outro.



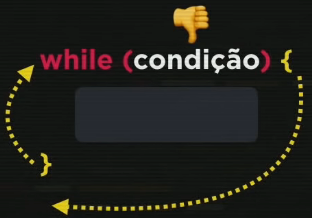
**Repetições (laços):** Um laço começa como uma condição e testa uma expressão: ou esse teste é verdadeiro ou falso. Mas o comportamento é diferente: **se a condição for verdade**, não vai partir para o bloco seguinte, ele vai voltar, **vai fazer um *looping*, um laço, uma repetição**. Voltando, ele vai verificar se a condição ainda é verdadeira. Sendo, ele vai voltar e fazer outro laço. Ele **vai fazer isso enquanto a condição for verdadeira**. A partir do momento que o losango não for verdade, o laço é quebrado e o resto do comando é executado.



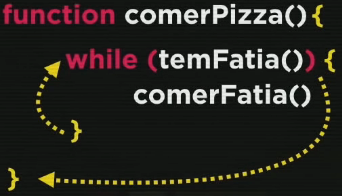
**Escrevendo a estrutura WHILE (*ENQUANTO)* em JavaScript**

Escrevo *while* e ponho, entre parênteses, a condição que eu testaria naquele losango. Quando coloco um *while*, tenho que colocar um bloco, que são sinais de chaves.

Então, esse bloco que está relacionado a esse enquanto, vai ser executado enquanto essa condição for verdadeira. Depois que for executado, chegando na chave debaixo, vai voltar para o enquanto e continuar o teste. Quando passa a ser falsa, o fluxo é desviado para o lado de fora.



Montando a estrutura da Pizza com *While*:



Enquanto tem fatia, eu vou criar um bloco e dentro desse bloco eu vou colocar o comando *comerFatia( )*. **IMPORTANTE**: Nesse caso, não importa o tamanho da pizza, que o programa servirá. Já no caso sequencial, só pode ser usado para se eu tiver uma pizza de 4 fatias.

**Exemplo While:**

*// console.log("Tudo bem?");*

*// console.log("Tudo bem?");*

*// console.log("Tudo bem?");*

*var* c = 1

while (c <= 3) {

console.log("Tudo bem?")

c++ *//3 + 1*

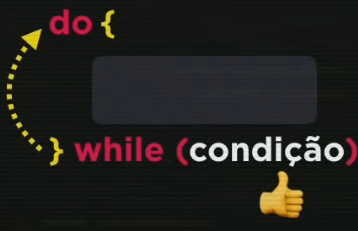
}

*//Enquanto meu contador for menor ou igual a 3, vou escrever tudo bem e incrementar.* *Por exemplo, C é menor que 1? Vai voltar, vai escrever e vai somar mais um.* *E vai voltando até C ser menor ou igual que 3.*

Sendo assim, **a estrutura de repetição *While* é classificada como estrutura de repetição com teste lógico no início**, pois ele faz o teste e sendo verdadeiro, ele faz o bloco (teste, executa, faz looping).

Só que existe a possibilidade de fazer a estrutura ao contrário: primeiro eu executo o bloco e depois eu faço o teste. Se o teste for verdadeiro, faz o looping e executa o bloco. E vai ficar nesse looping e executa o bloco de novo. E vai ficar nesse looping até ser verdade. E quando não for verdade, sai do bloco.

**Estrutura de Repetição *DO...WHILE (FAÇA... ENQUANTO)***

******

Em vez de *while* alguma coisa, escreve “***do***” (faça), abre e fecha o bloco, faça esse bloco, enquanto a condição for verdadeira. E quando for falsa, sai do bloco e segue a execução normal do programa.

**Comparando teste lógico no início e no fim**

|  |  |
| --- | --- |
| C:\Users\Angélica\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\9F2B7262.tmp  C:\Users\Angélica\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\A4BF04CE.tmp | C:\Users\Angélica\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\9F2B7262.tmp  C:\Users\Angélica\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\F78067FA.tmp |

São essencialmente estruturas de repetição, mas o comportamento é diferente. O bloco de cima primeiro testa e depois executa o bloco. A de baixo, primeiro executa o bloco e depois testa.

*/\* While*

*var c = 1*

*while (c <= 3) {*

*console.log(`Passo ${c}`)*

*c++ //3 + 1*

*}*

*\*/*

*//Do While*

*var* c = 1

do {

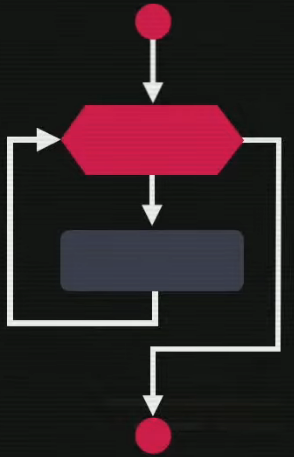
console.log(**`**Passo ${c}**`**)

c++

} while (c <= 3)

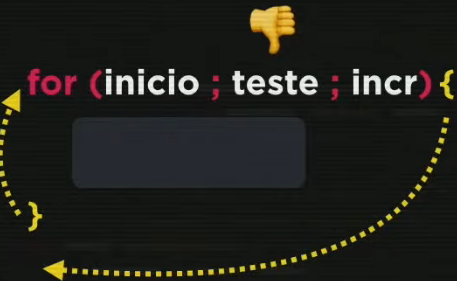
**Repetições (Parte 2) - Curso JavaScript #14**

**Estrutura de repetição com variável de controle (*for*)**: representada pelo hexágono regular, onde, basicamente, são feitas 3 coisas: inicialização, teste lógico e incremento.



Na primeira passagem, faz a inicialização e o teste lógico. Se o teste lógico for verdadeiro, ele vai executar o bloco (que pode ter um ou vários comandos) e vai voltar, mas, no momento do looping, vai também fazer o incremento. A cada vez que for verdadeiro e voltar, o incremento é feito. E quando o teste for falso, segue o caminho de encerramento.

**Escrevendo o FOR no JavaScript**



A estrutura for tem três partes: a inicialização, o teste e o incremento, como dito anteriormente. E além disso, tem um bloco.

Na primeira passagem pelo *for*, faz-se a inicialização e o teste; sendo o teste verdadeiro, faz-se a execução do bloco (cuja quantidade de comando pode ser inclusive, dentro do for, um *while*, *switch*, *do while*...) e uma vez executado, vai voltar pra cima. Só que enquanto tá voltando, vai fazer o incremento e volta. E isso vai acontecer enquanto for verdadeiro.

**Comparando *While* e *For***

|  |  |
| --- | --- |
| C:\Users\Angélica\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\A4BF04CE.tmp  C:\Users\Angélica\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\E79184E4.tmp | C:\Users\Angélica\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\B8CC1092.tmp  C:\Users\Angélica\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\12B4CE70.tmp |

While:

*var* c = 1

while (c <= 5) {

console.log(c)

c++

}

Do While:

*var* 1

do {

console.log(c)

c++

} while (c <= 5)

For:

for (*var* c = 1; c <= 5; c++){

console.log(c)

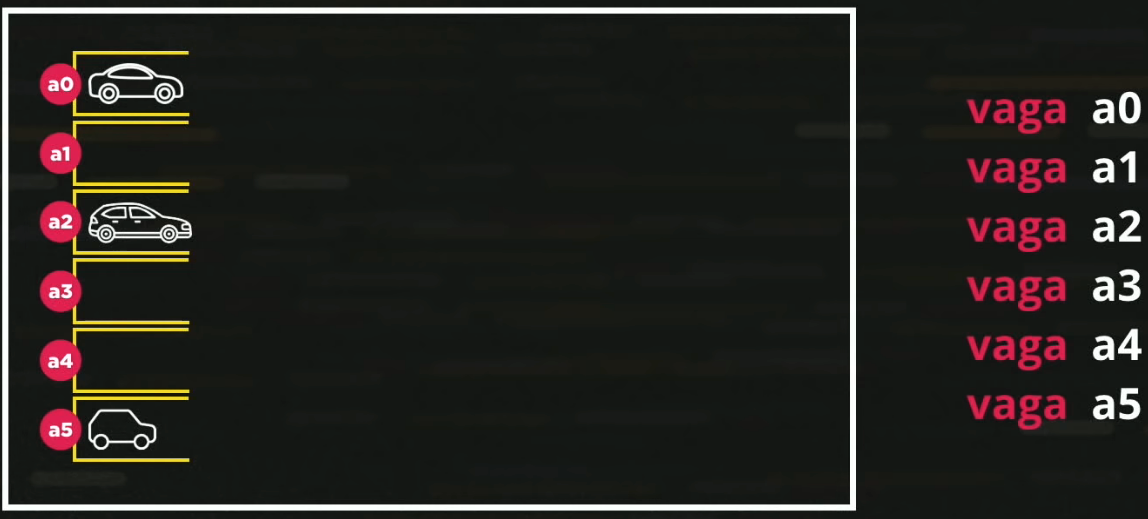
};

**Variáveis Compostas (Arrays) - Curso JavaScript #15**

**Variáveis simples só conseguem armazenar um valor por vez**. Por exemplo, se criei uma variável *n* com *let* ou *var* que recebe 5, se eu tentar atribuir 8 ela vai perder o 5, pois ela é uma variável simples.

**Variáveis compostas devem ser capazes de armazenar vários valores em uma mesma estrutura**. Isto é, se eu colocar *var n = 5*, ela guarda. Se eu colocar 8, também. Se eu colocar 12, também. Ela não vai perder os valores anteriores para armazenar os novos valores.

Quanto a gente precisa de muitas variáveis num programa, elas consomem muito código. Por exemplo:

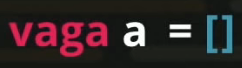


Se neste exemplo eu tiver um carro na vaga 2 e quiser colocar outro carro nessa mesma vaga, eu vou ter que tirar esse carro que já está na vaga.

Para melhorar esse gerenciamento, não vou mais dar um identificador para cada uma das vagas. Vou chama-las todas pelo mesmo nome. Todas vão se chamar *a* (automóveis). E vou colocar um identificador de índice, uma chave que vai identificar cada uma delas. E para declarar essas vagas, só vou usar uma linha *vaga a* e para cada vaga eu coloco uma chave, um índice.



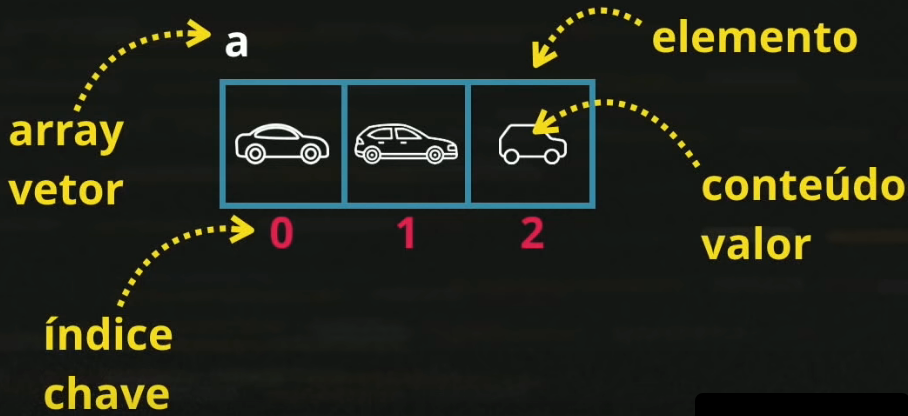
Para que eu declare isso como uma variável composta, preciso utilizar colchetes: *vaga a = [ ]* :



Pra declarar uma variável desse tipo e já colocar os carros na vaga, faço:

C:\Users\Angélica\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\D23CF308.tmp

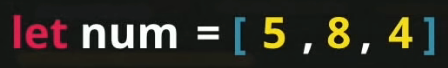
Quando eu coloco os dados ali, ele já vai arrumando os carros na vaga. Basicamente, o que vai acontecer é que vou declarar uma variável a e quando eu faço a linha anterior, automaticamente os três carros vão para os espaços abaixo e cada espaço recebe um número, um índice.

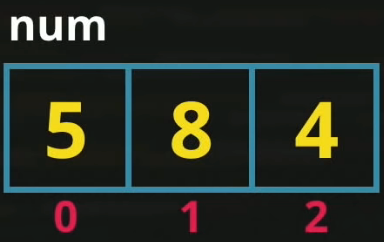


Essa variável *a* é do tipo composta ou um ***array***, em português, **vetor**. O vetor é uma variável com vários espacinhos. Ele é **composto de elementos**. Um elemento de um vetor é um par que **agrupa o espaço da memória, o valor colocado dentro dele e o índice**. O índice é o número abaixo, que pode ser chamado de **chave**. Por último, está o **conteúdo de cada elemento**.

Um array ou um vetor ou uma variável composta é uma variável que tem vários elementos. Cada elemento é composto por seu valor e uma chave de identificação.

**Transformando as vagas para uma linguagem JavaScript:**

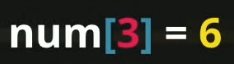




**Métodos internos de vetores**

* **[ ]**

Se eu quiser acrescentar a vaga 3, ou seja, mais um espaço, é só fazer:



Ou seja, “coloque o valor 6 na posição 3”. O JavaScript percebe que essa linha não existe e cria mais um elemento, coloca o incide 3 e coloca o valor 6.

* ***.push(valor)***

Se eu quiser que o JavaScript ponha na última posição, preciso usar .*push(valor)*:

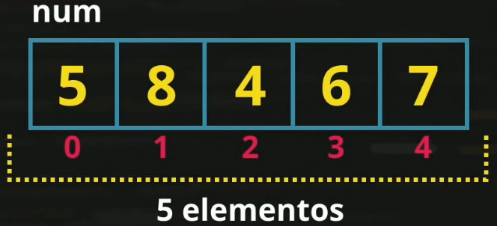
C:\Users\Angélica\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\15B2F0AC.tmp

Com isso, o JavaScript cria mais um elemento e percebe que a chave é a 4 e o valor é 7.

* ***.length***

Para eu saber o comprimento de um *array*, utilizo *.length*. **E é importante dizer que não tem parênteses no JavaScript, pois não é um método é um atributo**:

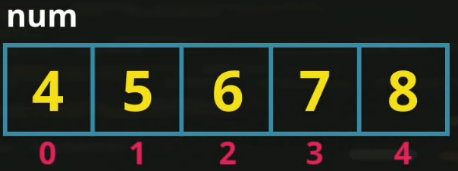
C:\Users\Angélica\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\BB58179A.tmp



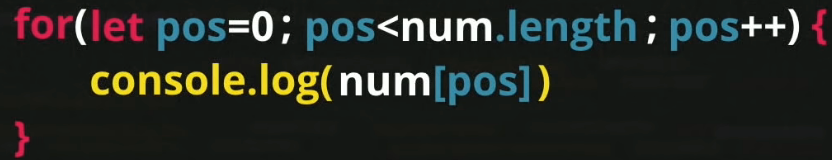
* ***.sort()***

Quando utilizo o método *.sort( )*, ele pega todos os elementos e coloca em ordem crescente:

C:\Users\Angélica\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\F2962186.tmp

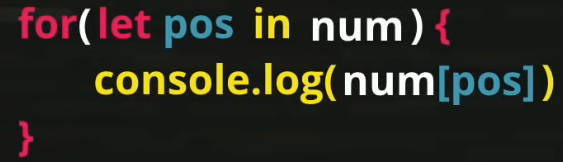


Pra escrever um vetor sem colchetes:



A partir de 0, enquanto não chegar no final do meu vetor, eu adiciono 1 cada vez que fizer um *looping*.

Esse é o código tradicional do percurso de vetores. Mas o JavaScript tem um jeito mais fácil, mais simplificado, otimizado para variáveis compostas e que funciona só para *arrays* e objetos (já que todo array em JavScript é um object):



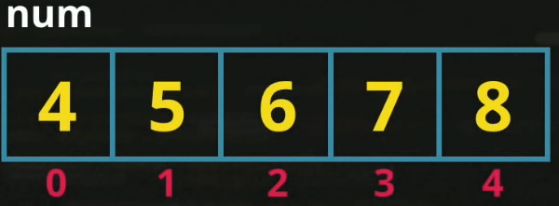
Primeiramente, ponho meu índice e do lado do *in* ponho o nome da variável composta. Então, para cada posição dentro de *num*, exiba num[pos].

**Buscando valores dentro de um vetor**

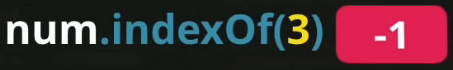
Vou utilizar o método *index****O****f():*

C:\Users\Angélica\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\5A84FB9E.tmp

Por exemplo, com esse comando, pesquiso se tem o valor 7 e o JavaScript responde “sim, esse valor existe e está na chave 3”.



Caso eu pesquise pelo valor 3, que não existe, o JavaScript retorna o valor -1, que para ele, diz que pesquisou dentro do vetor e não encontrou nenhuma ocorrência.



**Funções - Curso JavaScript #16**

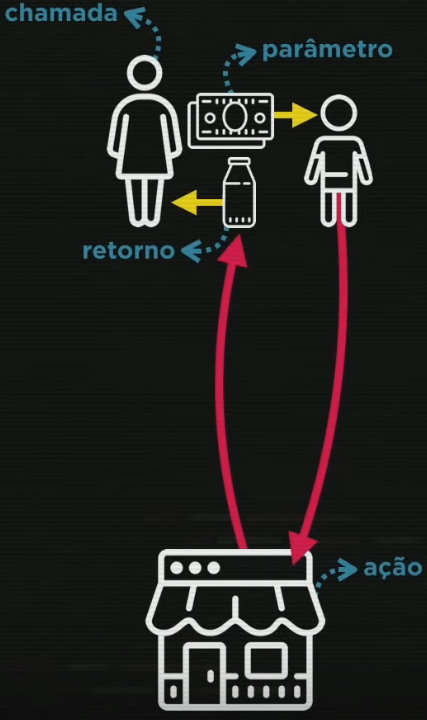
**JavaScript é uma linguagem funcional** e, por isso, **funções** são tão importantes. Vamos abstrair o conceito.

Sua mãe te dá várias funções, seja varrer o chão ou arrumar o quarto, estas são tarefas/rotinas que você tem que fazer.

A mãe chega para o filho, entrega um dinheiro toda semana e pede pra ele ir ao mercado comprar leite. O filho vai pegar a garrafa de leite e vai entregar para a mãe. E uma vez que sua funcionalidade terminou, ela encerrou.

Para a mãe, não importa como você chegou ao mercado, quanto foi o leite e como o pegou. O que importa é que ela dê o dinheiro, passe a ordem (quero leite) e a missão do filho é trazer um leite, vendo apenas o resultado.

**Fatores de uma função**

****

**Chamada**: toda função precisa ter uma chamada. Às vezes é a mãe chamando para comprar leite ou, às vezes, uma funcionalidade que aconteça através de um relógio, como, toda quarta-feira de manhã tenho que ir comprar o leite.

**Parâmetro**: você não pode ir comprar o leite sem algo de entrada, sem o parâmetro. Nesse caso, existem dois: o dinheiro que a mãe vai dar e o que a mãe vai pedir pra ser feito com esse dinheiro, comprar leite.

**Ação**: Quando o filho pega o dinheiro e vai, da maneira que for, ao mercado, o processo todo que acontece para ele comprar o leite.

**Retorno**: Quando o filho volta para casa e entrega o leite pedido pela mãe.

ATENÇÃO:

Nem toda função tem parâmetros, nem toda função tem retornos, mas uma função só pode ter um retorno.

**Definição técnica de Funções**

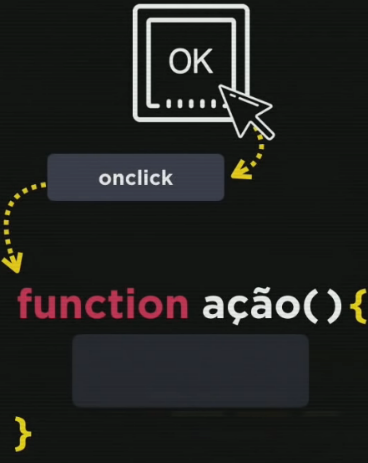
Funções são **ações** executadas assim que são **chamadas** ou em decorrência de um **evento**.

Ou seja, você vai executar alguma coisa quando sua mãe te chamar ou quando acontecer algum evento (“Todo dia que chover, tem que tirar roupa do varal”).

Uma **função** pode receber **parâmetros** e retornar um **resultado**.

Parâmetros, no caso, é o dinheiro. E o resultado, no caso, é a garrafa de leite.

**Relembrando...**

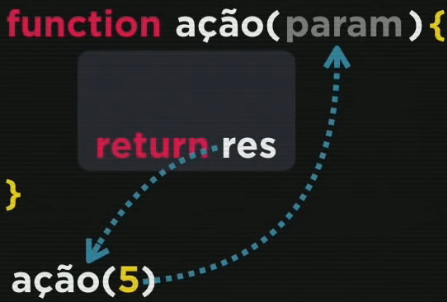


**Upgrade no conceito de função**

Dentro dos parênteses, passar parâmetros que vão ser processados pela ação e podem ter um resultado. Essa ação não executa por padrão, ela tem que ter uma chamada, no caso, ação(5) –– estou chamando a ação (que é o nome da função) e, nessa linha, estou fazendo a chamada.

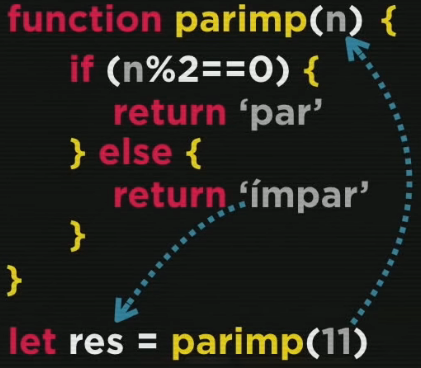
Na hora que vou executar esse código, ele vai pular a parte da função e vai executar somente esse código ação(5), passando um valor 5, que para o parâmetro, que valerá 5 nesse exemplo.

Vou poder fazer o processamento do bloco e, no fim, poderei dar um retorno de um resultado, que vai ser voltado para quem fez a chamada (a mãe, na história antiga).

**Exemplo de funcionalidade: Par ou Ímpar?**

Tenho uma função chamada *parimp*, passando n como parâmetro. Dentro dessas chaves, vou verificar se o *n* é *pa*r ou *ímpar*, verificando se ele é divisível por 2.



Criei uma variável chamada *res*, chamei a função e passei o parâmetro *11*, que vai retornar o *else*. Quando retornar *ímpar*, essa palavra fica dentro de *res*.

