

CONFIGURAÇÃO DE AMBIENTE

- Node: ambiente de execução JavaScript
- NPM: gerenciador de pacotes do Node

JS: FUNDAMENTOS

1) ALGORITMOS

É uma sequência de passos, podendo ter repetições e condições.



Ex.: Como saio de casa e chego ao shopping?



Algoritmos podem ser representados como:

- Fluxogramas
- Linguagem natural (português, inglês, etc)
- Linguagem artificial (java, C++, PHP, etc)
- Pseudo-línguagem

2) ESTRUTURA DE DADOS

Visa trazer ordem e estrutura para os dados.

- Ex.: Lista de aprovados | Pilha de livros
- Fila do banco | Árvore de pastas

Estruturas de dados básicas

- inteiros
- reais
- letras
- bool (V ou F)

Dados são armazenados em variáveis e constantes, sendo usados em operações (atribuições, aritméticas, relacionais, lógicas).

3) ORGANIZAÇÃO BÁSICA DE UM CÓDIGO JS

- Não é necessária a utilização de ";" como em outras linguagens de programação.
- Textos podem ser colocados entre aspas simples ou duplas.
- Blocos de código são colocados entre chaves.

4) EXECUTANDO JS

É possível executar códigos JS de forma online em sites como www.jsfiddle.net ou nas ferramentas de desenvolvedor dos navegadores. Nesse último caso, apenas as bibliotecas usadas no site estarão disponíveis no console.

Além disso, também é possível executar em IDE's e alguns editores de texto, como o VS Code, por exemplo, bem como no terminal, através do Node.

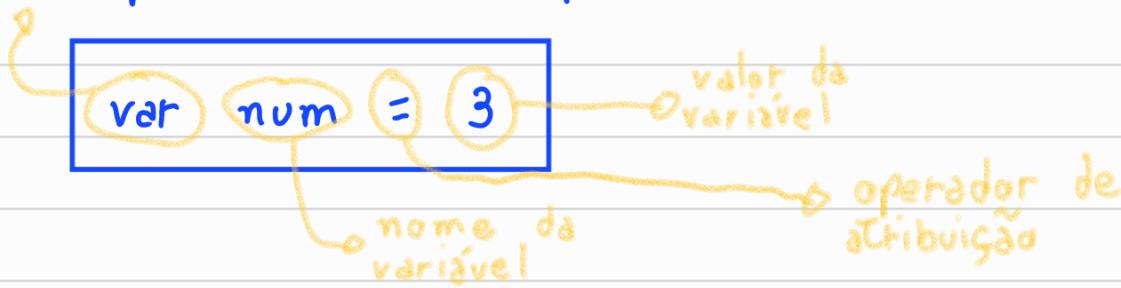
Assim como em outras linguagens, os comentários dentro do código não são executados.

↳ Comentários de linha única começam com "//", enquanto que blocos de comentários ficam entre "/*" e "*/".

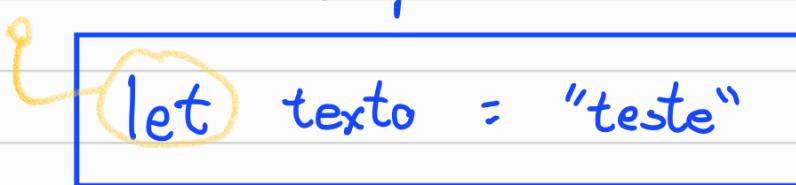
↳ Sempre evitar comentários óbvios ou desnecessários.

S) VAR, LET E CONST

a) var: palavra reservada para declarar variáveis no JS.

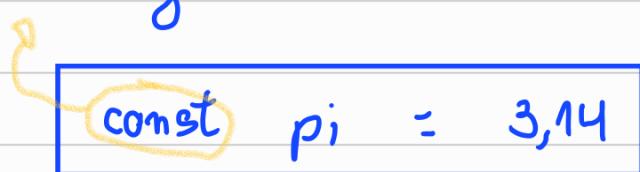


b) let: forma mais moderna de criar variáveis, sendo mais preferível do que o uso de var.



Uma variável declarada com let não pode ser redeclarada. Já as declaradas com var podem ser redeclaradas normalmente,

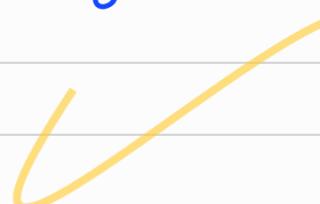
c) const: usada para declarar constantes, isto é, valores que, após declarados, não podem ser alterados em outro ponto do código.



6) TIPAGEM

O JavaScript é uma linguagem de tipagem dinâmica (fraca), então uma mesma variável pode possuir tipos diferentes de valores ao longo do código.

let qualquer = "Texto"
qualquer = 3.14



7) TIPOS EM JAVASCRIPT

a) number: valores numéricos, podendo ser números inteiros ou não.

```
const peso1 = 1.0  
const peso2 = Number("2.0")  
console.log(Number.isInteger(peso1))
```

→ converte texto em número
→ testa se a variável é um inteiro

```
const prova1 = 9.871  
const prova2 = 6.871
```

```
const total = prova1 * peso1 + prova2 * peso2  
const media = total / (peso1 + peso2)  
console.log(media.toFixed(2))
```

→ limita o número de casas decimais

Observações:

- $10 / 0 = \text{Infinity}$
- "10"/2 = 5
- "show" * 2 = NaN (not a number)
- $0.1 + 0.7 = 0,7\bar{9}$

Algumas funções matemáticas necessitam do uso do objeto Math para serem calculadas, como exponenciação, por exemplo.

```
const pi = Math.PI  
const exp = Math.pow(7, 2)
```

→ expoente
→ base

b) strings: é uma cadeia de caracteres, que são delimitadas por aspas simples ou duplas.

```
const escola = "aula"  
console.log(escola.charAt(2))
```

→ obtém o caractere que está na posição especificada

```
console.log(escola.indexOf('u'))
```

→ obtém o índice do caractere especificado

Observação:

→ O primeiro caractere de uma string está no índice 0.

A função `substring()` obtém parte de uma string.

A função `replace()` substitui um valor dentro da string por outro.

A função `split()` divide uma string com base no delimitador especificado.

É possível concatenar 2 strings com o sinal "+" ou com a função `concat()`.

Para interpolar uma string com os valores de variáveis, é necessário utilizar um recurso chamado de Template String.

Uma Template String é delimitada por crases e as variáveis que serão inseridas devem ser delimitadas por um par de chaves antecedido por "\$".

```
const nome = "Rebeca"  
const template = `Olá, ${nome}!`
```

As template strings também reconhecem quebras de linha sem a necessidade de informar o código de quebra de linha. Também é possível fazer a interpolação de expressões e funções.

```
console.log(`1+1 = ${1+1}`)  
const escola = "aula"  
console.log(`${escola.toUpperCase()}`)
```

c) boolean: em JavaScript, o tipo boolean pode ter valores diferentes de true e false, como 0 e 1, por exemplo.

```
let isAtivo = true  
console.log(isAtivo)
```

```
isAtivo = false  
console.log(isAtivo)
```

```
isAtivo = 0  
console.log (!!isAtivo)
```

o true

o false

o necessário para retornar um valor booleano, que, no caso, será false

Observações:

- Valores numéricos diferentes de zero, listas vazias, strings não vazias e infinity são considerados true;
- Strings vazias, null, NaN e undefined são considerados false.

d) arrays: é uma estrutura unidimensional usada para agrupar vários valores de forma linear a partir de um único identifi-

cador. Array (ou Vetor) é uma estrutura indexada, em que cada valor contido dentro dele pode ser acessado através de um índice numérico, iniciando pelo índice 0.

O array em JavaScript é heterogêneo, então aceita agrupar valores de tipos diferentes, e de tamanho dinâmico

```
const valores = [7.7, 8.9, 6.3, 9.2] → representado por um par de colchetes
console.log(valores[0], valores[3]) → 7.7 e 9.2
console.log(valores[4]) → undefined
```

O tamanho de um array pode ser obtido por meio da propriedade `length`. Além disso, novos valores podem ser adicionados no final do array com a função `push()`, consequentemente aumentando o tamanho do array.

```
valores.length → 4
valores.push(null, "teste", true)
valores.length → 7
```

Também é possível remover valores de um array com a função `pop()` ou com a palavra reservada `delete`.

```
valores.pop() → remove o último valor
delete valores[0] → remove uma posição específica
```

e) object: diferente de um array que é representado por um par de colchetes, os objetos são representados por um par de chaves. Um objeto é uma coleção de chaves e valores. É possível declarar um objeto vazio para posteriormente atribuir chaves e valores, pois objetos são elementos dinâmicos.

```
const prod1 = {}  
prod1.nome = "celular"  
prod1.preco = 4999.90  
console.log(prod1)
```

⇒ { nome: 'celular', preco: 4999.90 }

Os objetos também podem ter atributos de nome composto se declarados entre colchetes e aspas, mas não é uma boa prática.

```
prod1["Desconto legal"] = 0.40  
console.log(prod1)
```

⇒ { nome: 'celular', preco: 4999.9,
'Desconto legal': 0.4 }

Objetos podem ser criados de forma direta ao informar suas propriedades entre as chaves. Também é possível que um objeto possua outro objeto.

```
const prod2 = {  
    nome: "camisa",  
    preco: 79.90,  
    obj: {  
        num: 1  
    }  
}  
console.log(prod2)
```

⇒ { nome: 'camisa', preco: 79.9,
obj: { num: 1 } }

Utiliza-se o ":" para acessar os elementos de um objeto, sejam eles funções, propriedades, etc.

8) NULL & UNDEFINED

A diferença entre null e undefined é que null seria a ausência de um valor, enquanto que o undefined é a

não definição de um valor.

```
let valor // não inicializada  
console.log(valor) → undefined  
valor = null // ausência de valor  
console.log(valor) → null
```

Quando o objetivo for limpar uma variável que aponta para um objeto ou para uma função, basta atribuir null.

É recomendado deixar apenas a própria linguagem fazer atribuição de undefined quando ela julgar necessário.

9) FUNÇÃO (BÁSICO)

Função é um bloco de código que possui um nome e que pode ou não retornar alguma informação específica após ser executada. As funções também podem ou não receber dadas de entrada (parâmetros).

```
o palavra reservada  
function imprimirSoma(a, b){  
    console.log(a + b)  
}  
  
imprimirSoma(2, 3) → 2 + 3 = 5  
imprimirSoma(2) → 2 + undefined = NaN  
imprimirSoma(2, 10, 4, 5) → 2 + 10 = 12  
imprimirSoma() → o NaN
```

Para fazer com que a função retorne uma infor-

maçõo, é preciso usar a palavra reservada return.

```
function soma (a,b = 0){  
    return a+b  
}
```

console.log(soma(2,3))

console.log(soma(2))

parâmetro opcional

$$2+3=5$$

$$2+0=2$$

Constantes e variáveis podem armazenar funções também, como funções anônimas (funções sem nome), por exemplo:

```
const imprimirSoma = function(a,b){  
    console.log(a+b)  
}
```

imprimirSoma(2,3)

$$2+3=5$$

Há também funções chamadas de "arrow", onde a assinatura e o corpo da função são separados por " $=>$ ". Esse tipo de função também pode ser armazenado em variáveis e constantes:

```
const soma = (a,b) => {  
    return a+b  
}
```

console.log(soma(2,3))

$$2+3=5$$

A arrow function permite que as funções sejam escritas de forma ainda mais resumida:

const subtração = (a, b) => a - b

console.log(subtração(2, 3))

const imprimir = a => console.log(a)
imprimir('legal')

retorno implícito

2 - 3 = -1

parâmetro
único

legal

10) LET VS VAR

Variáveis declarados com a palavra reservada "let" podem possuir 3 escopos: global, função e bloco. Já as variáveis declaradas com "var" não possuem o escopo de bloco:

var num = 1
{

var num = 2

console.log(num) → 2

}

console.log(num) → 2

ignora o bloco em que está e
sobre escreve o valor da variável

let num2 = 1
{

let num2 = 2

console.log(num2) → 2

}

console.log(num2) → 1

substitui o valor apenas
dentro do bloco em que está

Esse diferença entre os escopos causa diferenças em outras situações também, como em um laço de repetição:

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



```
for(let x=0; x<10; x++) {  
    console.log(x)  
}  
  
console.log(x) → x is not defined
```

i é global

x não existe fora do bloco do for

Um problema histórico da palavra "var" em laços de repetição é o seguinte:

```
const func = []  
for(var i = 0; i < 10; i++) {  
    func.push(function () {  
        console.log(i)  
    })  
}  
  
func[2]() → 10  
func[8]() → 10
```

Na prática, não é possível saber qual era o valor de *i* no momento em que a função anônima foi armazenada na respectiva posição do vetor. Se a variável "*i*" tivesse sido

declarado com "let", os resultados obtidos ao chamar "Func[2]()" e "Func[8]()" seriam "2" e "8", respectivamente.

Outra diferença entre "var" e "let" é o conceito de hoisting. O hoisting nada mais é do que o interpretador passar as declarações da variáveis para o topo do código. Entretanto, o hoisting ocorre apenas quando a declaração é feita com "var".

11) OPERADORES

a) atribuição: o uso de um único sinal de igual atribui o valor da direita no elemento da esquerda. Também é possível juntar o operador de atribuição com operadores aritméticos.

<code>const a = 7</code>	$b = b + a$	$3 + 7 = 10$
<code>let b = 3</code>	$b = b - 4$	$10 - 4 = 6$
	$b *= 2$	$6 * 2 = 12$
	$b /= 2$	$12 / 2 = 6$
	$b \% = 2$	(resto da divisão de b por 2)
	<code>console.log(b)</code>	$\leftarrow 0$

b) destructuring: extrai elementos de um objeto.

```
const pessoa = {  
    nome: 'Ana',  
    idade: 5,  
    endereco: {  
        logradouro: 'Rua ABC',  
        numero: 1000  
    }  
}
```

```
const {nome, idade} = pessoa  
console.log(nome, idade)
```

```
const {nome: n, idade: i} = pessoa  
console.log(n, i)
```

cria 2 constantes
contendo as informações do objeto

Ana 5

cria as constantes "n" e "i" contendo os valores dos atributos "nome" e "idade" do objeto, respectivamente

Ana 5

Quando o elemento que se está tentando extrair não existe, a variável gerada é considerada como undefined. Entretanto, também é possível definir um valor padrão para esses casos.

undefined true

```
const {sobrenome, bemHumorado = true} = pessoa  
console.log(sobrenome, bemHumorado)
```

Para obter valores dentro de um objeto que está dentro de outro objeto o destructuring deve ser escrito da seguinte forma:

```
const {endereço: {logradouro}} = pessoa  
console.log(logradouro) → Rua ABC
```

```
const {conta: {agencia}} = pessoa
```

Erro, pois não é possível obter a propriedade "agência" de um objeto undefined ou null, como "conta", que não existe dentro de "pessoa".

Também é possível extrair elementos de arrays usando destructuring.

```
const [x] = [10]
```

```
console.log(x) → 10
```

```
const [n1, , n3, , n5, n6 = 0] = [10, 7, 9, 8]
```

```
console.log(n1, n3, n5, n6)
```

10 9 undefined 0

Em caso de array de arrays, o destructuring funciona da seguinte forma:

```
const [[, nota]] = [[8, 8], [9, 6, 8]]  
console.log(nota) → 6
```

Ao utilizar destructuring, é necessário pensar também na legibilidade do código. No exemplo acima, o ideal talvez fosse acessar o array direto pelo índice.

```
function rand({min = 0, max = 1000}) {  
    return Math.random() * (max - min) + min  
}
```

```
const obj = {  
    max: 500,  
    min: 100  
}
```

```
console.log(rand(obj))
```

Exibe um número entre 100 e 500

c) aritméticos:

```
const [a,b,c,d] = [3,5,1,15]
```

```
const soma = a + b + c + d  
const subtracao = a - b - c - d  
const multiplicacao = b * c  
const divisao = c / d  
const modulo = b % 2
```

d) relacionais: comparam valores e expressões, sempre resultando em verdadeiro ou falso.

```
console.log('1' == 1) → true  
console.log('1' === 1) → False  
console.log('3' != 3) → False  
console.log('3' !== 3) → true
```

```
console.log(3 < 2) → true  
console.log(3 <= 2) → false  
console.log(3 <= 2) → false  
console.log(3 >= 2) → true
```

```
console.log(undefined == null) → true  
console.log(undefined === null) → false
```

Os operadores relacionais são:

- igual (não compara os tipos): ==
- estritamente igual (compara os tipos): ===
- maior: >
- menor: <
- maior ou igual: >=
- menor ou igual: <=

e) lógicos: formam expressões lógicas que resultam em verdadeiro ou falso.

Os operadores lógicos são:

- E: &&
- OU: ||
- OU EXCLUSIVO: não possui um símbolo específico em JavaScript
- NEGAÇÃO: ! (operador unário)

Os resultados sempre seguem a definição da tabela-verdade:

$$V \text{ e } V = V$$

$$F \text{ e } ? = F$$

$$V \text{ ou } ? = V$$

$$F \text{ ou } F = F$$

$$V \text{ xor } V = F$$

$$!V = F$$

$$V \text{ xor } F = V$$

$$!F = V$$

$$F \text{ xor } F = F$$

Funários: são operadores que possuem apenas um operando, como por exemplo operadores de negação, incremento e decremento.

Alguns operadores unários podem ser tanto pré-fixados quanto pós-fixados, por exemplo:

```
let num1 = 1
```

```
let num2 = 2
```

`num1++` → pós-fixado

`console.log(num1)` → 2

`--num1` → pré-fixado

`console.log(num1)` → 1

`console.log(++num1 == num2--)` → true, pois no momento da comparação ambos valem 2

g) ternário: utiliza 3 operandos, onde a primeira parte da expressão usa um operador relacional, a segunda e a terceira definem consequências caso verdadeiro e consequências caso falso.

```
const resultado = nota => nota >= 7 ? "Aprovado" : "Reprovado"
```

```
console.log(resultado(7.1))
```

```
console.log(resultado(6.9))
```

→ Aprovado

→ Reprovado

O símbolo "?" é usado para definir a consequência caso verdadeiro, enquanto que o símbolo ":" é usado para definir caso falso.

12) TRATAMENTO DE ERRO

Alguns blocos de código possuem comandos que podem ocasionar erros de acordo com o contexto em que se encontram. Para esses casos, é necessário usar estruturas específicas para tratamento de erros:

```
function tratamento(erro){  
    throw {  
        name: erro.name,  
        msg: erro.message  
    }  
}
```

→ lança o erro

```
function imprimirNome(obj){  
    try {  
        console.log(obj.name.toUpperCase())  
    } catch (e) {  
        tratamento(e)  
    } finally {  
        console.log("final")  
    }  
}
```

→ tenta executar o bloco

→ executa caso haja erro no bloco try

→ executa independente de ter ou não ocorrido erro no bloco try

```
const obj = {name: "Roberto"}  
imprimirNome(obj)
```