Apostila de JavaScript

Sumário

Aula 1

Cod1

```
//Execução
// terminal :
//node index.js

//declarando constantes
const nome = "Jose";
const idade = 27;
// aspas simples e duplas têm o mesmo efeito
const sexo = "M";
const endereco = 'Rua K, 12'

console.log(nome, sexo, idade, endereco)

//declarando variáveis
//let: variável local com escopo de bloco
let a = 2;
let b = "abc";
//var: seu escopo é a função em que foi declarada ou global
var c = 2 + 3;
var d = "abcd"
console.log(a, b, c, d)
//ponto e virgula é opcional
b = 20
console.log(b)//possibilidade de mudar o tipo da variavel

/*
var: variavel global ou com escopo dentro da função onde foi criada
*/
/*let fica dentro do escopo */
x = "Valor fora do escopo"
if (a > 1) {
    let x = "Agora estou dentro";
    var y = "Sou Global"
    a = "Sou global"// a é variavel global definida anteriormente
portanto a alteração permanece
    console.log("dentro da chave", x, y);

}

console.log("Y existe:", y)
console.log("X não existe fora do escopo:", x)
```

```
console.log("a é variavel global definida anteriormente portanto a alteração permanece:", a);
```

Output:

```
Jose M 27 Rua K, 12

2 abc 5 abcd

20

dentro da chave Agora estou dentro Sou Global

Y existe: Sou Global

X não existe fora do escopo: Valor fora do escopo

a é variavel global definida anteriormente portanto a alteração permanece: Sou global
```

COD 2

```
console.log("Concatenando textos podemos concatenar com o simbolo + ou concatenar fazendo uma F-Strings utilizando `texto ${variavel} texto` \nExemplos")

var linguagem = "Javascript";
console.log("Aprendendo " + linguagem);
//nome pode ser redeclarada
var linguagem = "Java";
console.log("Aprendendo, " + linguagem);
//escopo não restrito a bloco
var idade = 18;
//exibe undefined. Ou seja, a variável já existe aqui, só não teve valor atribuído.
//Ela é içada - do inglês hoist - para fora do bloco if
console.log(`Oi, ${nome}`);
if (idade >= 18) {
    var nome = "João";
    console.log(`Parabéns, ${nome}. Você existe Globalmente`);
}
//ainda existe aqui
console.log(`Até mais, ${nome} global.`);//obs usar acento grave para fazer uma string assim

console.log('Usando " não funciona ${nome} -- intepreta como string')
```

```
console.log("Usando ' não funciona ${nome} -- intepreta como
string")
console.log("_____________________________________________________
________")

const n1 = 2;
const n2 = '3';
//coerção implícita de n1, concatenação acontece
const n3 = n1 + n2;
console.log(`${n1} + ${n2} = ${n3}`);
//coeração explícita, soma acontece
const n4 = n1 + Number(n2)
console.log(`${n1} + ${n2} = ${n4}`)

/*
console.log(1 == 1)//true
console.log (1 == "1") //true //coerção implicita -- coloca os dois
do mesmo tipo e compara
console.log (1 === 1) //true
console.log (1 === "1")//false // comparação tipo com tipo valor
com valor //melhor usar esse habitualmente
console.log (true == 1) //true
console.log (1 == [1])//true
console.log (null == null)//true
console.log (null == undefined)//true
console.log([] == false)//true
console.log ([] == [])//false

*/
console.log("_____________Comparações_________________")
console.log(`1 == 1 --> ${1 == 1}`)//true
console.log(`1 == "1" --> ${1 == "1"}`) //true //coerção implicita
-- coloca os dois do mesmo tipo e compara
console.log(1 === 1) //true
console.log(1 === "1")//false // comparação tipo com tipo valor com
valor //melhor usar esse habitualmente
console.log(true == 1) //true
console.log(true == 2)//false
console.log(true == -1)//false
console.log(false == 0)//True

console.log(1 == [1])//true
console.log(null == null)//true
console.log(null == undefined)//true
console.log([] == false)//true
console.log([] == [])//false
```

Output:

```
Concatenando textos podemos concatenar com o simbolo + ou
concatenar fazendo uma F-Strings utilizando `texto ${variavel}
texto`

Exemplos

Aprendendo Javascript

Aprendendo, Java

Oi, undefined

Parabéns, João. Você existe Globalmente

Até mais, João global.

Usando " não funciona ${nome} -- intepreta como string

Usando ' não funciona ${nome} -- intepreta como string

_____________________________________________________________

2 + 3 = 23

2 + 3 = 5

_____________Comparações_________________

1 == 1 --> true

1 == "1" --> true

true

false

true

false

false

true

true

true

true

true

false

_____________________________________
```

Aula 2

Cod1

```
//declaração
v1 = [];
//podemos acessar qualquer posição, começando de zero
v1[0] = 3.4;
v1[10] = 2;
v1[2] = "abc"
//aqui, v1 tem comprimento igual a 11
console.log(v1.length)
//inicializando na declaração
v2 = [2, "abc", true]
console.log(v2)
//iterando
for (let i = 0; i < v2.length; i++){
    console.log(v2[i])
}

for (let i = 0; i <v1.length; i++){
    console.log("Elemento "+ i + " : " + v1[i])
}

console.log("_____
_____")

const nomes = ["Ana Maria", "Antonio", "Rodrigo",
"Alex","Cristina"];
console.log(nomes)
console.log("Filtros (verificar em
todos_____
_____")

const apenasComA = nomes.filter((n) => n.startsWith("A")); //filtro
|| criterio de seleção começa com A || Função lambida em python ou
arrow em java scrit : (n) => n.startsWith("A") || Pega todo o vetor
e aplica o filtro nele se der true
console.log(apenasComA);

const apenasComR = nomes.filter(n => n.startsWith("R"));

console.log(apenasComR);
console.log("Aplicar a
todos_____
_____")
```

```
///map equivale ao all no python
const res = nomes.map((nome) => nome.charAt(0));//map aplica em
todos os elementos o dados
console.log(res);

const todosComecamComA = nomes.every((n) =>
n.startsWith("A"));//Todos os elementos da lista começãm com a?
true ou false no caso false
console.log(todosComecamComA);

console.log("_____________________________________________________
____________________")

const valores = [1, 2, 3, 4];
console.log(valores)
const somaValores = valores.reduce((ac, v) => ac + v);//reduz o
vetor a um unico elemento -- de acordo com a regra
/* o que ele fez?

    1+2 = 3 - colocar na primeira posição
    1º posição = 3 + 3(segunda posição) = 6
    6(1º posição) + 4(segunda posição) = 10//(resposta final 10)
   output = 10
*/
console.log("Reduce_____________________________________________
_____________________")

console.log(somaValores);

console.log("Funções_____________________________________________
_____________________")

function hello(){
    console.log ('Oi')
}
hello()
//cuidado, aqui redefinimos a função sem parâmetros
/* */ // comente e descomente essa função
function hello (nome){//função anterior deixou de existir
    console.log ('Hello, ' + nome)
}
/* */
hello('Pedro')

function soma (a, b) {
    return a + b;
}
const somando = soma (2, 3)
```

```
console.log (somando)

//Funções anonimas || sem nome
console.log("Funções
Anonimas_____________________________________________________
________")


const dobro = function (n) {
    return n * 2;
};
const resposta = dobro(4);
console.log(resposta);
//valor padrão para o parâmetro
const triplo = function (n = 5) {
    return 3 * n;
};
console.log(triplo());
console.log(triplo(10));

//_____________________________________________
console.log("Arrow
Functions____________________________________________________
_________")
/*
Há um símbolo => - daí o nome arrow - entre eles. Uma arrow
function
não tem nome e também pode ser armazenada em constantes ou
variáveis. Além
disso, arrow functions têm as seguintes características.

❮ Quando a lista de parâmetros possui um único argumento, os
parênteses podem ser omitidos.
 Quando o corpo possui uma única instrução, as chaves podem ser
omitidas.
 Quando o corpo possui uma única instrução que produz um valor a
ser devolvido, a instrução return é opcional: Se usar as chaves,
deve-se usar o return. Caso contrário, ele não pode ser usado.
*/
const oi = () => console.log("Hello");

oi();
const dobro2 = (valor) => valor * 2;
console.log(dobro2(10));
const triplo2 = (valor) => {
    return valor * 3;
};
console.log(triplo2(10));
//e agora?
```

```
const ehPar = (n) => {
    return n % 2 === 0;
};
console.log(ehPar(10));//recebo o resultado da função como
argumento

console.log(ehPar);//recebo a função como argumento


/*

Primeiro, em Javascript, funcões são cidadãs de primeira
classe. Informalmente, um cidadão de primeira classe em uma
linguagem de
programação é uma entidade que oferece suporte a operações como as
seguintes.
( Ser passada como argumento para uma função.
( Ser devolvida por uma função.
( Ser atribuída a uma variável.

 Há também o conceito de função
de alta ordem. Uma função de alta ordem é aquela que recebe pelo
menos
uma função como parâmetro e/ou devolve uma função quando seu
processamento
termina.
 precisa de outra função para funcionar -> como parametro ex map,
reduce, filtros

*/









//NOTAS ALEATORIAS
//python os vetores usam a biblioteca numpy - vetores mais
flexiveis

//programação funcional - sem usar um loop explicito -> reduz
numero de linhas de codigo -> uso um mecanismo da linguagem para
fazer um loop para mim
```

Output:

```
11
[ 2, 'abc', true ]
2
abc
true
Elemento 0 : 3.4
Elemento 1 : undefined
Elemento 2 : abc
Elemento 3 : undefined
Elemento 4 : undefined
Elemento 5 : undefined
Elemento 6 : undefined
Elemento 7 : undefined
Elemento 8 : undefined
Elemento 9 : undefined
Elemento 10 : 2
_____
[ 'Ana Maria', 'Antonio', 'Rodrigo', 'Alex', 'Cristina' ]
Filtros (verificar em
todos_____
[ 'Ana Maria', 'Antonio', 'Alex' ]
[ 'Rodrigo' ]
Aplicar a
todos_____
[ 'A', 'A', 'R', 'A', 'C' ]
false
_____
[ 1, 2, 3, 4 ]
Reduce_____
10
Funções_____
```

```
Hello, undefined

Hello, Pedro

5

Funções
Anonimas_____________________________________________

8

15

30

Arrow
Functions_____________________________________________
_____________

Hello

20

30

true

[Function: ehPar]
```

Cod2

```
console.log("Clouser _____________________________")

/*uma função pode ser atribuída a uma variável*/
let umaFuncao = function () {
    console.log("Fui armazenada em uma variável");
}
umaFuncao()
//
_____________________________________________________________
_________

function f(funcao) {//f recebe uma função como parâmetro e, por
isso é uma função de alta ordem
    funcao()//chamando a função note como a tipagem dinâmica tem
seu preço
}
function g() {//g devolve uma função, portanto também é de alta
ordem.
    function outraFuncao() {
        console.log("Fui criada e estou dentro por g");
    }
    return outraFuncao;
```

```
}

f(function () {//f pode ser chamada assim -- com criação de uma
função anonima
    console.log('Sou uma função anonima e estou sendo passada para
f')
    26
})
console.log("\nA função g pode ser chamada: ")
const gResult = g()
console.log("Chamada 1 gResult = g(); gResult()\n")
gResult()

console.log(gResult)


console.log("\nChamada 2 g()()\n")
g()()//chama g --> g() retorna outrafunção || segunda chaves
executa a outra função outrafunção

console.log(g)
console.log(g())

console.log("Misturando f e g")

console.log(f(g))
f(g)//f chama g, que somente devolve uma função. Nada é exibido.


console.log(f(g()))//executa a função e não possui return
f(g())//f chama a função devolvida por g.


/*
//f tenta chamar o que a função criada por g devolve. Ela não
devolve coisa alguma. Por isso, um erro - somente em tempo de
execução - acontece. */
/*
f(g()()) dá ruim f nn recebe uma função como parametro

f(1)
*/
```

Output

```
Clouser _____________________________
Fui armazenada em uma variável
Sou uma função anonima e estou sendo passada para f
```

A função g pode ser chamada:

Chamada 1 gResult = g(); gResult()

Fui criada e estou dentro por g

[Function: outraFuncao]

Chamada 2 g()()

Fui criada e estou dentro por g

[Function: g]

[Function: outraFuncao]

Misturando f e g

undefined

Fui criada e estou dentro por g

undefined

Fui criada e estou dentro por g

```
Clouser ______________________________
Fui armazenada em uma variável
Sou uma função anonima e estou sendo passada para f

A função g pode ser chamada:
Chamada 1 gResult = g(); gResult()

Fui criada e estou dentro por g
[Function: outraFuncao]

Chamada 2 g()()

Fui criada e estou dentro por g
[Function: g]
[Function: outraFuncao]
Misturando f e g
undefined
Fui criada e estou dentro por g
undefined
Fui criada e estou dentro por g
```

Aula 3

Cod1

```
function
ola() {
    let nome
= 'João';
    return
function () {
        console.log('Olá, João');
```

```
    }
    6
}

let olaResult = ola();
/*perceba que aqui a função ola já terminou.
    10 É de se esperar que a variável nome já não
    11 possa ser acessada.*/
olaResult();

//também vale com parâmetros
function saudacoesFactory(saudacao, nome) {
    return function () {
        console.log(saudacao + ', ' + nome);

    }

}
let olaAna = saudacoesFactory('Olá', 'Ana');
let tchauHelena = saudacoesFactory('Tchau', 'Helena');
olaAna();
tchauHelena();
```

Output:

```
Olá, João

Olá, Ana

Tchau, Helena
```

Cod2

```
function eAgora() {
    let cont = 1;
    function f1() {
        console.log(cont);
    }
    cont++;
    function f2() {
        console.log(cont);
    }
    //JSON contendo as duas funções
    return { f1, f2 }//funções chamadas depois do incremento --> ou
seja as funções são executadas apenas agora(depois do incremento do
cont)
}
```

```
let eAgoraResult = eAgora();

/* neste momento, a funcao eAgora já
executou por completo e a variável
cont já foi incrementada. Seu valor final
é mantido e, assim, ambas f1 e f2 exibirão 2.
*/
eAgoraResult.f1();
eAgoraResult.f2();
```

Output:

```
2
2
```

Json

```
//  JSON - Javascript Object Notation


let pessoa = {
    nome: "João",
    idade: 17,
}

console.log("Me chamo " + pessoa.nome);//o acesso a propriedades
pode ser feito com ponto
console.log("Tenho " + pessoa["idade"] + " anos");//e com [] também



let pessoaComEndereco = {
    nome: "Maria",
    idade: 21,
    endereco: {
        logradouro: "Rua B",
        numero: 121,
    },
};
console.log(
    `Sou ${pessoaComEndereco.nome},
    tenho ${pessoaComEndereco.idade} anos
    e moro na rua ${pessoaComEndereco.endereco["logradouro"]}
    número ${pessoaComEndereco["endereco"]["numero"]}.
    Gosto muito de morar no endereço $
{pessoaComEndereco.endereco.logradouro} - $
{pessoaComEndereco.endereco.numero}`
```

```
);

//console.log(pessoaComEndereco.endereco.logradouro + " - " +
pessoaComEndereco.endereco.numero)
```

Output:

```
Me chamo João
Tenho 17 anos
Sou Maria,
   tenho 21 anos
   e moro na rua Rua B
   número 121.
   Gosto muito de morar no endereço Rua B - 121
```

Estoque

```
let concessionaria = {
    cnpj: "00011122210001-45",
    endereco: {
        logradouro: "Rua A",
        numero: 10,
        bairro: "Vila J",
    },
    veiculos: [
        {
            marca: "Ford",
            modelo: "Ecosport",
            anoDeFabricacao: 2018,
        },
        {
            marca: "Chevrolet",
            modelo: "Onix",
            anoDeFabricacao: 2020,
        },
        {
            marca: "Volkswagen",
            modelo: "Nivus",
            anoDeFabricacao: 2020,
        },
    ],
};
for (let veiculo of concessionaria.veiculos) {
    console.log(`Marca: ${veiculo.marca}`);
    console.log(`Modelo: ${veiculo.modelo}`);
```

```
    console.log(`Ano de Fabricação:
    ${veiculo.anoDeFabricacao}`);
}
```

Output :

```
Marca: Ford
Modelo: Ecosport
Ano de Fabricação:
  2018
Marca: Chevrolet
Modelo: Onix
Ano de Fabricação:
  2020
Marca: Volkswagen
Modelo: Nivus
Ano de Fabricação:
  2020
```

Calculadora

```
let calculadora = {
    //pode ser arrow function
    soma: (a, b) => a + b,
    //e função comum também
    subtracao: function (a, b) {
        return a - b;
    },
};
console.log(`2 + 3 = ${calculadora.soma(2, 3)}`);
console.log(`2 - 3 = ${calculadora.subtracao(2, 3)}`);
```

Output:

```
2 + 3 = 5
2 - 3 = -1
```

Sincrona

```
//Execução Síncrona e Assíncrona
//Sincrona 1 espera a resposta do outro ex conversa
// Assicrona email mando o email e nn fico esperando a resposta sem
fazer outra coisa -- eu mando o email e de tempos em tempos eu
confiro se eu recebi


// 3.1 Modelo Single Threaded Ambientes de execução Javascript são
Single
// Threaded. Isso quer dizer que há um único fluxo de execução. Não
há execução de código em paralelo. Como mostra o Bloco de Código
3.1.1, as instruções
// são executadas uma após a outra, na ordem em que foram
definidas. Não há a
// possibidade de uma instrução i executar antes de outra instrução
j(∀i > j).


console.log("Sou sincrono ou ____________________________________-")
console.log('Eu primeiro')
console.log("Agora eu")
console.log("Conta longa: Operação i : 2^100 : " + 2 ** 100)
console.log("mesmo sendo menor que i eu j nn vou ser executada
antes")

console.log("Sempre vou ser a última...:(")
console.log("Eu estou executando na ordem mesmo que um processo
seja mais lento que o outro (eu espero o processo finalizar para
exe)")

console.log("_____________________________________________________
___")
function demorada() {
    const atualMais2Segundos = new Date().getTime() + 2000
    //não esqueça do ;, única instrução no corpo do while
    while (new Date().getTime() <= atualMais2Segundos);
    const d = 8 + 4
    return d
}
const a = 2 + 3
const b = 5 + 9
const d = demorada()
/*
o valor de e não depende do valor devolvido
pela função demorada.
*/
```

```
const e = 2 + a + b
console.log("o valor de e não depende do valor devolvido pela
função demorada porém ele espera por ela para exectar. e = " + e)
```

Output :

```
Sou sincrono ou _____-
Eu primeiro
Agora eu
Conta longa: Operação i : 2^100 : 1.2676506002282294e+30
mesmo sendo menor que i eu j nn vou ser executada antes
Sempre vou ser a última...:(
Eu estou executando na ordem mesmo que um processo seja mais lento que o
outro (eu espero o processo finalizar para exe)
_____
o valor de e não depende do valor devolvido pela função demorada porém ele
espera por ela para exectar. e = 21
```

Assincrona

```
console.log("Assincrona_____")

function demorada() {
    const atualMais2Segundos = new Date().getTime() + 2000
    //não esqueça do ;, única instrução no corpo do while
    while (new Date().getTime() <= atualMais2Segundos);
    const d = 8 + 4
    return d
}
const a = 2 + 3
const b = 5 + 9
//função será executada depois de, pelo menos, 500 milissegundos
setTimeout(function () {
    const d = demorada()
    console.log("eu sou demorada(vou depois) :" + d)
}, 500)//tempo espera para dar inicio 500

//enquanto isso, essas linhas prosseguem executando
//sem ficar esperando
```

```
const e = a + b
console.log("sou rapido(vou primeiro):" + e)

console.log("Enfileiramento_____________________________________________
_____")

setTimeout(function () {
    console.log('dentro da timeout')
}, 0)
const d = new Date().getTime() + 1000
//não esqueça do ;, única instrução no corpo do while
while (new Date().getTime() <= d);
console.log('fora da timeout')
```

Output :

```
Assincrona_____________________________________
sou rapido(vou primeiro):19
Enfileiramento_____________________________________________
fora da timeout
dentro da timeout
eu sou demorada(vou depois) :12
```

Aula 4

Cod1: Assincronas

```
setTimeout(function () {
    console.log('dentro da timeout', 0)
})
const a = new Date().getTime() + 1000
//não esqueça do ;, única instrução no corpo do while
while (new Date().getTime() <= a);
console.log('fora da timeout')
```

Output:

```
fora da timeout
```

```
dentro da timeout 0
```

Thread

```
const fs = require("fs");
const abrirArquivo = function (nomeArquivo) {
    const exibirConteudo = function (erro, conteudo) {
        if (erro) {
            console.log(`Deu erro: ${erro}`);
        } else {
            console.log(conteudo.toString());
        }
    };
    fs.readFile(nomeArquivo, exibirConteudo);//quando o readFIle
terminar chama o exibir conteudo callbacks
};
//crie um arquivo chamado arquivo.txt com o conteúdo``2'' (sem as
aspas)
//no mesmo diretório em que se encontra seu script
abrirArquivo("arquivo.txt");
```

Output

```
olá thead nova

+ arquivo txt (olá thead nova)
```

Promise

```
/*Uma Promise é um objeto por meio do qual uma função pode propagar
um resultado ou um erro em algum momento no futuro.


 Quando uma promise é produzida e o processamento associado a ela
ainda não está concluído, ela está no estado Pending.
❮ Quando o processamento associado a uma promise termina com
sucesso, ela passa para o estado Fullfilled.
❮ Quano o processamento associado a uma promise termina com erro,
ela passa
para o estado Rejected.
❮ Os estados Fullfilled e Rejected são estados finais. Uma vez que
uma promise se encontre em um desses estados, ela nunca transita
para outro estado.
❮ Uma promise pode ser criada em qualquer um dos três estados.
```

```
Uma das vantagens obtidas pelo uso de promises é a simplificação da
passagem
de parâmetros entre funções assíncronas. A sua execução pode ser
encadeada. //then ligar promises
*/



/* *q/
function calculoDemorado(numero) {
    return new Promise(function (resolve, reject) {//promessa de
execução --> executa paralelamente -- o then espera a outra
promisse acabar para executar o then se nn executa tudo
assincronamente
        let res = 0;
        for (let i = 1; i <= numero; i++) {
            res += i;
        }
        resolve(res);
    });
}
calculoDemorado(10).then((resultado) => {
    console.log(resultado)//soma dos numeros de 1 a 10
})

/* */
/* *q/
console.log("_____________________________________________________
_____________")

function calculoRapidinho(numero) {
    return Promise.resolve((numero * (numero + 1)) / 2);
}
calculoRapidinho(10).then(resultado => {
    console.log(resultado)
})
//Executa primeiro, mesmo que a promise já esteja fullfilled
console.log('Esperando...')


/* */

function calculoRapidinho(numero) {
    //comando ternario --> como se fosse if numero >=0
{promisse.resolve....} else {promisse reject}
    return numero >= 0
    ? Promise.resolve((numero * (numero + 1)) / 2)
    : Promise.reject("Somente valores positivos, por favor");
}
```

```
calculoRapidinho(10)
    .then((resultado) => {
    console.log('chamou o then do 10:',resultado);//chamado
    })
    .catch((err) => {
    console.log('n chamou o catch do 10:',err);
    });
    calculoRapidinho(-1)
    .then((resultado) => {
    console.log('n chamou o then do -1:',resultado);
    })
    .catch((err) => {
    console.log('chamou o catch do -1:',err);//chamado
    });
console.log("esperando...");//executa primeiro
```

Output:

```
esperando...
chamou o then do 10: 55
chamou o catch do -1: Somente valores positivos, por favor
```

Callbacks

```
const fs = require("fs");
const abrirArquivo = function (nomeArquivo) {
    console.log("Fui chamada")
    const exibirConteudo = function (erro, conteudo) {
        if (erro) {
            console.log(`Deu erro: ${erro}`);
        } else {
            console.log(conteudo.toString());
            const dobro = +conteudo.toString() * 2;
            //inicio finalizar
            const finalizar = function (erro) {
                if (erro) {
                    console.log('Deu erro tentando salvar o dobro')
                }
                else {
                    console.log("Salvou o dobro com sucesso");
                }
            }
            //fim finalizar
            fs.writeFile('dobro.txt', dobro.toString(),
finalizar);//callback finalizar
        }
    };
    fs.readFile(nomeArquivo, exibirConteudo);
};
abrirArquivo("dobro.txt");

//outro jeito de fazer codigo assincrono
```

Output :

```
Fui chamada
4
Salvou o dobro com sucesso
```

Aleatórios:

Constantes

```
const nome = "Jose";
const idade = 27;
// aspas simples e duplas têm o mesmo efeito
const sexo = "M";
const endereco = 'Rua K, 12'
```

console.log

```
console.log(nome , sexo,idade, endereco)
console.log("Concatenando textos podemos concatenar com o simbolo + ou
concatenar fazendo uma F-Strings utilizando `texto ${variavel} texto` \nExemplos")
```

```
var linguagem = "Javascript";
console.log("Aprendendo " + linguagem);
```

```
console.log(`Oi, ${nome}`);
```

Variáveis

```
//declarando variáveis
//let: variável local com escopo de bloco
let a = 2;
let b = "abc";
//var: seu escopo é a função em que foi declarada ou global
var c = 2 + 3;
var d = "abcd"
console.log(a,b,c,d)
```

```
//ponto e virgula é opcional
b = 20
console.log(b)//possibilidade de mudar o tipo da variável

/*
var: variavel global ou com escopo dentro da função onde foi criada */
/*let fica dentro do escopo */

x = "Valor fora do escopo"
if(a>1){
        let x = "Agora estou dentro";
        var y = "Sou Global"
        a = "Sou global"// a é variavel global definida anteriormente portanto a
alteração permanece
        console.log("dentro da chave",x,y);


}


console.log("Y existe:",y)
console.log("X não existe fora do escopo:",x)
console.log("a é variavel global definida anteriormente portanto a alteração
permanece:",a);


console.log("_____")
```

Tipos de Variáveis

```
const n1 = 2;
const n2 = '3';
//coerção implícita de n1, concatenação acontece
const n3 = n1 + n2;
console.log(`${n1} + ${n2} = ${n3}`);
//coeração explícita, soma acontece
const n4 = n1 + Number (n2)
console.log(`${n1} + ${n2} = ${n4}`)
```

Operações Lógicas

```
console.log(1 == 1)//true
console.log (1 == "1") //true //coerção implicita -- coloca os dois do mesmo tipo e compara
console.log (1 === 1) //true
console.log (1 === "1")//false // comparação tipo com tipo valor com valor //melhor usar esse habitualmente
console.log (true == 1) //true
console.log (1 == [1])//true
console.log (null == null)//true
console.log (null == undefined)//true
console.log([] == false)//true
console.log ([] == [])//false
```

Vetores

```
//declaração
v1 = [];
//podemos acessar qualquer posição, começando de zero
v1[0] = 3.4;
v1[10] = 2;
v1[2] = "abc"
//aqui, v1 tem comprimento igual a 11
console.log(v1.length)
//inicializando na declaração
v2 = [2, "abc", true]
console.log(v2)
//iterando
for (let i = 0; i < v2.length; i++) {
    console.log(v2[i])
}

for (let i = 0; i < v1.length; i++) {
    console.log("Elemento " + i + " : " + v1[i])
}
```

Filtros

```
const nomes = ["Ana Maria", "Antonio", "Rodrigo", "Alex", "Cristina"];
console.log(nomes)
```

```
console.log("Filtros (verificar em
todos_____________________________________________________________
_____")

const apenasComA = nomes.filter((n) => n.startsWith("A")); //filtro
|| criterio de seleção começa com A || Função lambida em python ou
arrow em java scrit : (n) => n.startsWith("A") || Pega todo o vetor
e aplica o filtro nele se der true
console.log(apenasComA);

const apenasComR = nomes.filter(n => n.startsWith("R"));

console.log(apenasComR);
console.log("Aplicar a
todos_____________________________________________________________
_____")

///map equivale ao all no python
const res = nomes.map((nome) => nome.charAt(0));//map aplica em
todos os elementos o dados
console.log(res);

const todosComecamComA = nomes.every((n) =>
n.startsWith("A"));//Todos os elementos da lista começãm com a?
true ou false no caso false
console.log(todosComecamComA);

console.log("_____________________________________________________
____________________")


const valores = [1, 2, 3, 4];
console.log(valores)
const somaValores = valores.reduce((ac, v) => ac + v);//reduz o
vetor a um unico elemento -- de acordo com a regra
/* o que ele fez?

1+2 = 3 - colocar na primeira posição
1º posição = 3 + 3(segunda posição) = 6
6(1º posição) + 4(segunda posição) = 10//(resposta final 10)
output = 10
*/
console.log("Reduce_____________________________________________________
_____________________")

console.log(somaValores);
```

Funções

```
console.log("Funções_____________________________________________________
_____________________")

function hello() {
    console.log('Oi')
}
hello()
//cuidado, aqui redefinimos a função sem parâmetros
/* */ // comente e descomente essa função
function hello(nome) {//função anterior deixou de existir
    console.log('Hello, ' + nome)
}
/* */
hello('Pedro')
function soma(a, b) {
    return a + b;
}
const somando = soma(2, 3)
console.log(somando)
```

Funções anônimas

```
//Funções anonimas || sem nome
console.log("Funções
Anonimas_____________________________________________________")



const dobro = function (n) {
return n * 2;
};
const resposta = dobro(4);
console.log(resposta);
//valor padrão para o parâmetro
const triplo = function (n = 5) {
return 3 * n;
};
console.log(triplo());
console.log(triplo(10));
```

```
//_____________________________________
console.log("Arrow
Functions_____________________________________________________________")
/*
Há um símbolo => - daí o nome arrow - entre eles. Uma arrow function
não tem nome e também pode ser armazenada em constantes ou variáveis. Além
disso, arrow functions têm as seguintes características.


❰ Quando a lista de parâmetros possui um único argumento, os parênteses podem
ser omitidos.
Quando o corpo possui uma única instrução, as chaves podem ser omitidas.
Quando o corpo possui uma única instrução que produz um valor a ser devolvido, a
instrução return é opcional: Se usar as chaves, deve-se usar o return. Caso contrário,
ele não pode ser usado.
*/
const oi = () => console.log("Hello");


oi();
const dobro2 = (valor) => valor * 2;
console.log(dobro2(10));
const triplo2 = (valor) => {
return valor * 3;
};
console.log(triplo2(10));
//e agora?
const ehPar = (n) => {
return n % 2 === 0;
};
console.log(ehPar(10));//recebo o resultado da função como argumento


console.log(ehPar);//recebo a função como argumento
```

Funções em formato de variável

```
console.log("Clouser _____________________________")

/*uma função pode ser atribuída a uma variável*/
let umaFuncao = function () {
    console.log("Fui armazenada em uma variável");
}
umaFuncao()
//
_____________________________________________________________________
__________
```

```
function f(funcao) {//f recebe uma função como parâmetro e, por isso é uma função de alta ordem
    funcao()//chamando a função note como a tipagem dinâmica tem seu preço
}
function g() {//g devolve uma função, portanto também é de alta ordem.
    function outraFuncao() {
        console.log("Fui criada e estou dentro por g");
    }
    return outraFuncao;
}

f(function () {//f pode ser chamada assim -- com criação de uma função anonima
    console.log('Sou uma função anonima e estou sendo passada para f')
    26
})
console.log("\nA função g pode ser chamada: ")
const gResult = g()
console.log("Chamada 1 gResult = g(); gResult()\n")
gResult()

console.log(gResult)


console.log("\nChamada 2 g()()\n")
g()()//chama g --> g() retorna outrafunção || segunda chaves executa a outra função outrafunção

console.log(g)
console.log(g())

console.log("Misturando f e g")

console.log(f(g))
f(g)//f chama g, que somente devolve uma função. Nada é exibido.


console.log(f(g()))//executa a função e não possui return
f(g())//f chama a função devolvida por g.


/*
//f tenta chamar o que a função criada por g devolve. Ela não devolve coisa alguma. Por isso, um erro - somente em tempo de execução - acontece. */
```

```
/*
f(g()()) dá ruim f nn recebe uma função como parametro

f(1)
*/
```

Funções dentro de funções -Cuidado com execução não linear

```
function eAgora() {
    let cont = 1;
    function f1() {
        console.log(cont);
    }
    cont++;
    function f2() {
        console.log(cont);
    }
    //JSON contendo as duas funções
    return { f1, f2 }//funções chamadas depois do incremento --> ou
seja as funções são executadas apenas agora(depois do incremento do
cont)
}

let eAgoraResult = eAgora();

/* neste momento, a funcao eAgora já
executou por completo e a variável
cont já foi incrementada. Seu valor final
é mantido e, assim, ambas f1 e f2 exibirão 2.
*/
eAgoraResult.f1();
eAgoraResult.f2();
EM AMBOS AS FUNÇÕES VÃO SER EXECUTADAS APÓS OS COMANDOS: let cont =
1; & cont++; ASSIM O CODIGO VAI RETORNAR EM AMBOS OS CASOS 2
```

Testes com funções

```
function ola() {
    let nome = 'João';
    return function () {
        console.log('Olá, João');
    }
    6
}

let olaResult = ola();
/*perceba que aqui a função ola já terminou.
```

```
    10 É de se esperar que a variável nome já não
    11 possa ser acessada.*/
olaResult();

//também vale com parâmetros
function saudacoesFactory(saudacao, nome) {
    return function () {
        console.log(saudacao + ', ' + nome);

    }

}
let olaAna = saudacoesFactory('Olá', 'Ana');
let tchauHelena = saudacoesFactory('Tchau', 'Helena');
olaAna();
tchauHelena();
```

Calculadora

```
let calculadora = {
    //pode ser arrow function
    soma: (a, b) => a + b,
    //e função comum também
    subtracao: function (a, b) {
        return a - b;
    },
};
console.log(`2 + 3 = ${calculadora.soma(2, 3)}`);
console.log(`2 - 3 = ${calculadora.subtracao(2, 3)}`);
```

Strings

```

```

```
const nome = "Jose";
// aspas simples e duplas têm o mesmo efeito
const sexo = "M";
```

json

```
//  JSON - Javascript Object Notation


let pessoa = {
    nome: "João",
    idade: 17,
}

console.log("Me chamo " + pessoa.nome);//o acesso a propriedades
pode ser feito com ponto
console.log("Tenho " + pessoa["idade"] + " anos");//e com [] também



let pessoaComEndereco = {
    nome: "Maria",
    idade: 21,
    endereco: {
        logradouro: "Rua B",
        numero: 121,
    },
};
console.log(
    `Sou ${pessoaComEndereco.nome},
    tenho ${pessoaComEndereco.idade} anos
    e moro na rua ${pessoaComEndereco.endereco["logradouro"]}
    número ${pessoaComEndereco["endereco"]["numero"]}.
    Gosto muito de morar no endereço $
{pessoaComEndereco.endereco.logradouro} - $
{pessoaComEndereco.endereco.numero}`
);

//console.log(pessoaComEndereco.endereco.logradouro + " - " +
pessoaComEndereco.endereco.numero)









let concessionaria = {
    cnpj: "00011122210001-45",
    endereco: {
        logradouro: "Rua A",
```

```
        numero: 10,
        bairro: "Vila J",
    },
    veiculos: [
        {
            marca: "Ford",
            modelo: "Ecosport",
            anoDeFabricacao: 2018,
        },
        {
            marca: "Chevrolet",
            modelo: "Onix",
            anoDeFabricacao: 2020,
        },
        {
            marca: "Volkswagen",
            modelo: "Nivus",
            anoDeFabricacao: 2020,
        },
    ],
};
for (let veiculo of concessionaria.veiculos) {
    console.log(`Marca: ${veiculo.marca}`);
    console.log(`Modelo: ${veiculo.modelo}`);
    console.log(`Ano de Fabricação:
    ${veiculo.anoDeFabricacao}`);
}
```

Execução Síncrona e Assíncrona

Sincrona

```
//Execução Síncrona e Assíncrona
//Sincrona 1 espera a resposta do outro ex conversa
// Assicrona email mando o email e nn fico esperando a resposta sem
fazer outra coisa -- eu mando o email e de tempos em tempos eu
confiro se eu recebi


// 3.1 Modelo Single Threaded Ambientes de execução Javascript são
Single
```

```
// Threaded. Isso quer dizer que há um único fluxo de execução. Não
há execução de código em paralelo. Como mostra o Bloco de Código
3.1.1, as instruções
// são executadas uma após a outra, na ordem em que foram
definidas. Não há a
// possibidade de uma instrução i executar antes de outra instrução
j(∀i > j).


console.log("Sou sincrono ou _____-")
console.log('Eu primeiro')
console.log("Agora eu")
console.log("Conta longa: Operação i : 2^100 : " + 2 ** 100)
console.log("mesmo sendo menor que i eu j nn vou ser executada
antes")

console.log("Sempre vou ser a última...:(")
console.log("Eu estou executando na ordem mesmo que um processo
seja mais lento que o outro (eu espero o processo finalizar para
exe)")

console.log("_____
___")
function demorada() {
    const atualMais2Segundos = new Date().getTime() + 2000
    //não esqueça do ;, única instrução no corpo do while
    while (new Date().getTime() <= atualMais2Segundos);
    const d = 8 + 4
    return d
}
const a = 2 + 3
const b = 5 + 9
const d = demorada()
/*
o valor de e não depende do valor devolvido
pela função demorada.
*/
const e = 2 + a + b
console.log("o valor de e não depende do valor devolvido pela
função demorada porém ele espera por ela para exectar. e = " + e)
```

Assincrona

```
console.log("Assincrona_____")

function demorada() {
```

```
    const atualMais2Segundos = new Date().getTime() + 2000
    //não esqueça do ;, única instrução no corpo do while
    while (new Date().getTime() <= atualMais2Segundos);
    const d = 8 + 4
    return d
}
const a = 2 + 3
const b = 5 + 9
//função será executada depois de, pelo menos, 500 milissegundos
setTimeout(function () {
    const d = demorada()
    console.log("eu sou demorada(vou depois) :" + d)
}, 500)//tempo espera para dar inicio 500

//enquanto isso, essas linhas prosseguem executando
//sem ficar esperando
const e = a + b
console.log("sou rapido(vou primeiro):" + e)

console.log("Enfileiramento_____________________________________
_____")

setTimeout(function () {
    console.log('dentro da timeout')
}, 0)
const d = new Date().getTime() + 1000
//não esqueça do ;, única instrução no corpo do while
while (new Date().getTime() <= d);
console.log('fora da timeout')









setTimeout(function () {
    console.log('dentro da timeout', 0)
})
const a = new Date().getTime() + 1000
//não esqueça do ;, única instrução no corpo do while
while (new Date().getTime() <= a);
console.log('fora da timeout')
```

Thread

```
const fs = require("fs");
const abrirArquivo = function (nomeArquivo) {
    const exibirConteudo = function (erro, conteudo) {
        if (erro) {
            console.log(`Deu erro: ${erro}`);
        } else {
            console.log(conteudo.toString());
        }
    };
    fs.readFile(nomeArquivo, exibirConteudo);//quando o readFIle
terminar chama o exibir conteudo callbacks
};
//crie um arquivo chamado arquivo.txt com o conteúdo``2'' (sem as
aspas)
//no mesmo diretório em que se encontra seu script
abrirArquivo("arquivo.txt");
```

Promise

```
/*Uma Promise é um objeto por meio do qual uma função pode propagar
um resultado ou um erro em algum momento no futuro.


 Quando uma promise é produzida e o processamento associado a ela
ainda não está concluído, ela está no estado Pending.
❮ Quando o processamento associado a uma promise termina com
sucesso, ela passa para o estado Fullfilled.
❮ Quano o processamento associado a uma promise termina com erro,
ela passa
para o estado Rejected.
❮ Os estados Fullfilled e Rejected são estados finais. Uma vez que
uma promise se encontre em um desses estados, ela nunca transita
para outro estado.
❮ Uma promise pode ser criada em qualquer um dos três estados.
```

```
Uma das vantagens obtidas pelo uso de promises é a simplificação da
passagem
de parâmetros entre funções assíncronas. A sua execução pode ser
encadeada. //then ligar promises
*/



/* *q/
function calculoDemorado(numero) {
    return new Promise(function (resolve, reject) {//promessa de
execução --> executa paralelamente -- o then espera a outra
promisse acabar para executar o then se nn executa tudo
assincronamente
        let res = 0;
        for (let i = 1; i <= numero; i++) {
            res += i;
        }
        resolve(res);
    });
}
calculoDemorado(10).then((resultado) => {
    console.log(resultado)//soma dos numeros de 1 a 10
})

/* */
/* *q/
console.log("_____________________________________________________
_____________")

function calculoRapidinho(numero) {
    return Promise.resolve((numero * (numero + 1)) / 2);
}
calculoRapidinho(10).then(resultado => {
    console.log(resultado)
})
//Executa primeiro, mesmo que a promise já esteja fullfilled
console.log('Esperando...')


/* */

function calculoRapidinho(numero) {
    //comando ternario --> como se fosse if numero >=0
{promisse.resolve....} else {promisse reject}
    return numero >= 0
    ? Promise.resolve((numero * (numero + 1)) / 2)
    : Promise.reject("Somente valores positivos, por favor");
    }
```

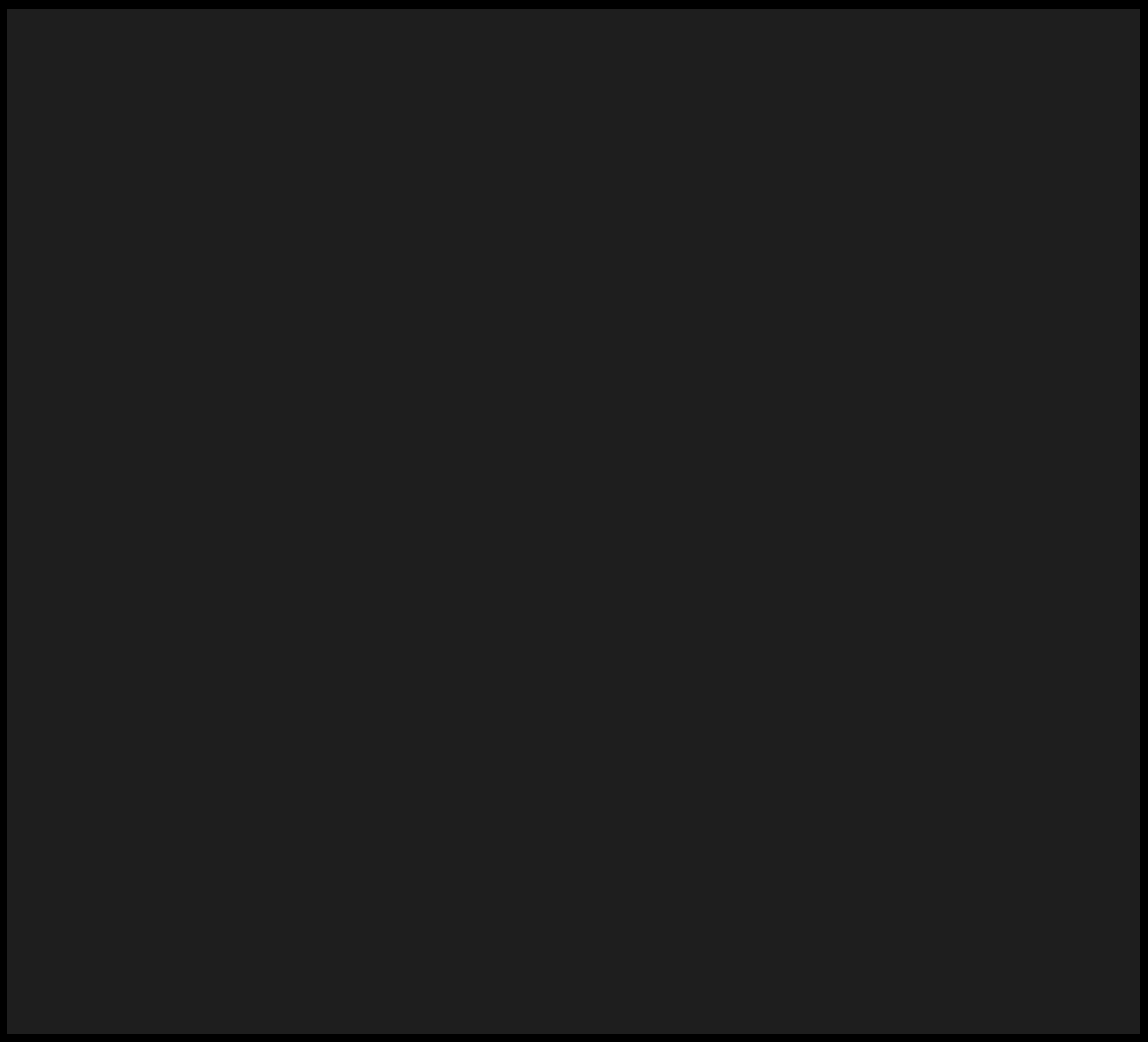
```
    calculoRapidinho(10)
    .then((resultado) => {
    console.log('chamou o then do 10:',resultado);//chamado
    })
    .catch((err) => {
    console.log('n chamou o catch do 10:',err);
    });
    calculoRapidinho(-1)
    .then((resultado) => {
    console.log('n chamou o then do -1:',resultado);
    })
    .catch((err) => {
    console.log('chamou o catch do -1:',err);//chamado
    });
    console.log("esperando...");//executa primeiro
```

Callbacks

```
const fs = require("fs");
const abrirArquivo = function (nomeArquivo) {
    console.log("Fui chamada")
    const exibirConteudo = function (erro, conteudo) {
        if (erro) {
            console.log(`Deu erro: ${erro}`);
        } else {
            console.log(conteudo.toString());
            const dobro = +conteudo.toString() * 2;
            //inicio finalizar
            const finalizar = function (erro) {
                if (erro) {
                    console.log('Deu erro tentando salvar o dobro')
                }
                else {
                    console.log("Salvou o dobro com sucesso");
                }
            }
            //fim finalizar
            fs.writeFile('dobro.txt', dobro.toString(),
finalizar);//callback finalizar
        }
```

```
    };
    fs.readFile(nomeArquivo, exibirConteudo);
};
abrirArquivo("dobro.txt");

//outro jeito de fazer codigo assincrono
```



Iniciado em	segunda, 10 abr 2023, 20:50
Estado	Finalizada
Concluída em	segunda, 10 abr 2023, 21:29
Tempo empregado	39 minutos 1 segundo
Avaliar	**7,00** de um máximo de 10,00(**70**%)

Questão 1

Correto Atingiu 1,00 de 1,00

Analise a figura e as proposições a seguir.

```
1  const f = () => {
2      console.log('f')
3  }
4
5  setTimeout(function (){
6      f()
7  })
8  console.log("p")
```

I. O script principal termina antes de a função agendada executar.

II. A função referenciada por f é chamada duas vezes.

III. A existência de instruções - console.log("p"), neste caso - depois da chamada a setTimeout causa um erro.

- I ✔
- II
- III
- I e III
- II e III

Sua resposta está correta.

A resposta correta é:
I.

Questão 2

Correto Atingiu 1,00 de 1,00

Considere a figura e as proposições a seguir.

```
1  var a = 2
2  var b = 2
3  let c = a + b
4  console.log(c)
5  b = 3
6  console.log(c)
```

I. Declara três variáveis.

II. Declara duas variáveis e uma constante.

III. A linha 3 causa um erro.

É correto apenas o que se afirma em

- I ✔
- II
- III
- I e III
- II e III

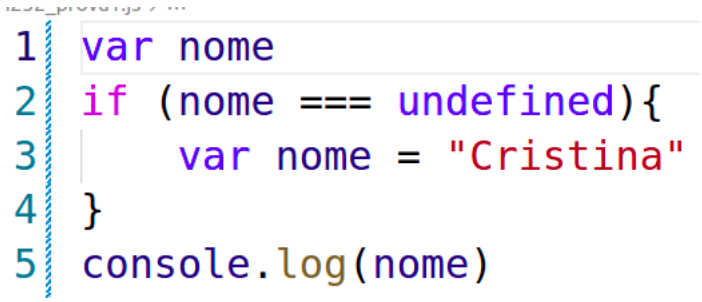
Sua resposta está correta.

A resposta correta é:
I.

Questão 3

Correto Atingiu 1,00 de 1,00

Analise a figure e as proposições a seguir.



```
1 var nome
2 if (nome === undefined){
3     var nome = "Cristina"
4 }
5 console.log(nome)
```

I. A linha 5 exibe undefined.

II. O resultado do teste realizado na linha 2 é true.

III. A linha 2 faz uma comparação apenas por tipo.

É correto apenas o que se afirma em

- I
- II ✔
- III
- I e II
- I e III

Sua resposta está correta.

A resposta correta é:
II.

Questão 4

Incorreto Atingiu 0,00 de 1,00

Analise a figura e as proposições a seguir.

```
1 const v = [100]
2 console.log(v.length)
3 v[2] = 5
4 console.log(v.length)
5 v = [5]
6 console.log(v.length)
```

I. A linha 1 cria um vetor de comprimento igual a 100.

II. A linha 3 causa um erro pois v é constante.

III. A linha 5 causa um erro pois v é constante.

É correto apenas o que se afirma em

- I
- II
- III
- I e II.
- II e III. ✖

Sua resposta está incorreta.

A resposta correta é:
III.

Questão 5

Correto Atingiu 1,00 de 1,00

Analise a figura e as proposições a seguir.

```
1 let valores = [1, 2, 3, 4, 5]
2 valores = valores.filter(v => v >= 3)
3 console.log(valores.length)
4 const res = valores.reduce((ac, v) =>{
5     return ac + v
6 })
7 console.log(res)
```

I. No exemplo, a função reduce produz um valor numérico em função da coleção chamada valores.

II. A função filter devolve uma coleção.

III. A linha 3 exibe undefined.

- I
- II
- III
- I e II. ✔
- II e III.

Sua resposta está correta.

A resposta correta é:
I e II..

Questão 6

Correto Atingiu 1,00 de 1,00

Analise a figura e as proposições a seguir.

```
1  const f1 = () => {
2      console.log('f1')
3  }
4  const f2 = (f) => {
5      console.log('f2')
6      f()
7      return f
8  }
9  f2(f1)()
```

I. A linha 6 causa um erro.

II. A função devolvida por f2 nunca é colocada em execução.

III. A linha 9 faz com que duas funções sejam colocadas em execução.

É correto apenas o que se afirma em

- I
- II
- III ✔
- I e II.
- II e III.

Sua resposta está correta.

A resposta correta é:
III.

Questão 7

Incorreto Atingiu 0,00 de 1,00

Considere as seguintes proposições a respeito de arrow functions.

I. ( ) => console.log('a') é uma arrow function válida.

II. function f ( ) {console.log('a')} é uma arrow function válida.

III. Dependendo de sua lista de parâmetros, uma arrow function pode omitir os parênteses que a delimitam (a lista).

É correto apenas o que se afirma em

- I
- I e II
- III ✖
- I e III
- II e III.

Sua resposta está incorreta.

A resposta correta é:
I e III.

Questão 8

Incorreto Atingiu 0,00 de 1,00

Considere a figura e as proposições a seguir.

```
1 const nome = "Pedro"
2 const idade = 22
3 const pessoa = {nome, idade}
4 console.log(pessoa.nome)
5 console.log(pessoa['idade'])
```

I. A linha 3 define um objeto JSON válido.

II. As linhas 4 e 5 são exemplos de acesso a propriedades JSON válidos.

III. Há pelo menos um erro no código.

É correto apenas o que se afirma em

- I
- II
- III
- I e II
- II e III ✖

Sua resposta está incorreta.

A resposta correta é:
I e II.

Questão 9

Correto Atingiu 1,00 de 1,00

Analise a figura e as proposições a seguir.

```
1  async function f1 (){
2      return 1
3  }
4  function f2(){
5      return Promise.resolve(1)
6  }
7  function f3(){
8      return Promise.reject(1)
9  }
10 async function teste(){
11     try{
12         const r1 = await f1()
13         console.log(r1)
14         const r2 = await f2()
15         console.log(r2)
16         const r3 = await f3()
17         console.log(r3)
18     }
19     catch (e){
20         console.log('e')
21     }
22 }
23 teste()
```

I. As funções f1, f2 e f3 podem usar a construção await em seu corpo, se necessário.

II. Depois da execução da linha 12, r1 passa a valer 1.

III. A saída do programa inclui a letra "e".

É correto apenas o que se afirma em

- I
- II
- III
- I e II
- II e III ✔

Sua resposta está correta.

A resposta correta é:
II e III.

Questão 10

Correto Atingiu 1,00 de 1,00

Analise a figura e as proposições a seguir.

```
1 console.log (2 === '2')
2 console.log(2 == '2')
```

I. As duas linhas produzem o mesmo resultado.

II. A linha 1 produz um erro.

III. A linha 2 faz coerção implícita.

É correto apenas o que se afirma em

- I
- II
- III ✔
- I e II
- II e III

Sua resposta está correta.

A resposta correta é:
III.

◄ Aula do Bossini

Seguir para...