O retorno seria o que?

Olha o exemplo:

```
function sum(a, b) {
        return a + b; // 2 e 5
}
```

Ele retorna a soma entre 2 + 5 que é 7. Por que o navegador está me falando que essa função **retorna**? Porque eu utilizei a palavra *return.*

Então eu posso fazer por exemplo:

let s1 = sum(2, 5); // Ele contém o valor que a função retornou -> 7

Mas olha esse segundo exemplo com o alert:

alert('Hello World!');

Após eu clicar no botão eu vou ver o retorno dessa função que é *undefined.* Mas porque essa função retornou isso? Porque a minha função não retornou nada.

Se eu faço como na função soma eu posso capturar o valor da soma em uma variável, posso usar esse valor direto na função, posso fazer algo assim:

```
function sum(a, b) {
    return a + b;
}

console.log(sum(3, 9));
```

A gente vê o valor 7 porque essa função retornou 7 e aí o console.log pegou o valor e exibiu na tela.

(O console.log está apenas exibindo o que eu estou pedindo pra ele exibir).

Mas se eu crio:

```
function sum2(a, b) {
    console.log(a + b); // Isso não é o retorno da função e sim o que ela está fazendo. Neste caso, ela é uma função inútil porque essa função não faz nada além de exibir alguma coisa no console. Mas pode ser que a função faça alguma coisa útil como por exemplo o alert ele não retorna nada mas faz alguma útil que é exibir alguma coisa na tela.
```

```
}

sum2(1, 2);
```

Uma outra coisa que a gente poderia ver que a gente já viu em aulas anteriories seria o exemplo de função que não retorna nada mas que faz alguma coisa útil seria:

```
document.addEventListener('click', function() {
    document.body.style.backgroundColor = 'brown';
});
```

Eu tenho uma função que não retorna nada porque eu não preciso do valor que essa função retorna, mas ela pode fazer alguma coisa útil como mudar a cor de fundo do body. Ela fez alguma ação, ela executou alguma coisa e eu não precisei do valor que essa função retorna.

Então a gente tem funções que retornam valor ou não retornam isso vai depender do contexto onde a gente está utilizando. Nos exercicíos que fizemos a gente fez algumas funções que não retornam valor mas que fazem alguma coisa e em outros nós fizemos funções que retornam o valor porque a gente precisava do valor retornado pela função. E eu posso colocar qualquer coisa no *return*:

```
function createPerson(name, middlename){
    return {
        name, middlename };
}
```

Eu poderia fazer assim também:

```
function createPerson(name, middlename){
    return {name: name, middlename: middlename }; // Chaves do objeto
}
```

Mas não é necessário porque como essas 2 variáveis tem o mesmo nome o JavaScript já assume que name será a chave name que tem o valor que vier nesse parâmetro e isso também serve para o middlename.

```
const p1 = criaPerson('Léo', 'Mancilha');
```

Essa variável contém um objeto e é a mesma coisa como se eu tivesse criado essa variável assim:

```
const p2 = {
```

```
    name: 'Léo',
    middlename: 'Mancilha'
};
```

As duas variáveis são iguais a única diferença é que agora eu tenho uma função que faz o trabalho para mim então eu não preciso ficar criando vários objetos eu posso usar a minha função para criar os objetos pra mim. O que muda são as maneiras que elas foram criadas.

A coisa começa a ficar mais complexa um pouco quando a gente faz alguma coisa assim por exemplo:

```
function sentence(start) {
    function rest(resto) {
        return resto; // Ele é o retorno do escopo da função rest
    }

    return rest;
}
```

return rest Ele retorna para a função sentence. Só que esta função retorna a função rest sem executar ela. E daí quando eu retorno uma função sem executar ela quer dizer que eu estou retornando a função em si. Então o uso disso vai ser um pouco diferente.

```
function sentence(start) {
    function rest(resto) {
        return start + '' + resto; // Ele é o retorno do escopo da
função rest
    }
    return rest; // Ele retorna para a função sentence. Só que esta
função retorna a função rest sem executar ela. E daí quando eu retorno
uma função sem executar ela quer dizer que eu estou retornando a função
em si.
}

const helloworld = sentence('Hi');
console.log(helloworld);
```

Agora eu estou usando dentro da função interna o parâmetro da função externa. Ele vai retornar: [Function: rest]

```
[Function: rest]
```

Essa função foi para a variável helloworld porque eu retornei a função rest. Então isso significa que helloworld é uma função então para eu executar uma função eu chamo os (), mas a função rest também recebe um parâmetro.

```
const helloworld = sentence('Hi ');
console.log(helloworld('world'));
```

Agora sim dentro da função rest eu estou retornando uma string;

Seria tipo isso também:

```
const helloworld = sentence('Hi ');
const resto = helloworld('world!');
console.log(resto);
```

Aqui a variável helloworld chama a função sentence só que a variável helloworld recebeu a minha função rest, então a função rest está na variável helloworld.

Isso pode ficar extremamente complexo porque por exemplo eu poderia ter mais uma função que retorna, fala outra coisa e aí o código vai ficando aquelas bonecas russas que você vai tirando uma de dentro da outra e aí lá no final tem uma bonequinha pequenininha que não fazia nada sem fazer nada era uma boneca imensa e você foi tirando e tirando e lá no dentro no final tinha uma bonequinha pequenininha.

Mas quando isso seria útil?

```
function duplica(n) {
    return n * 2;
}

function triplica(n) {
    return n * 3;
}

function quadriplica(n) {
    return n * 4;
}

console.log(duplica(2));
console.log(triplica(2));
console.log(quadriplica(2));
```

Meu código está repetido então se eu tivesse mais funções como 10 nesse sentido acima eu tenho que fazer mais funções que fazem esse serviço acima então claramente é a repetição que eu poderia ter evitado se eu utilizasse aquela tática que vimos:

```
function createMultiplicate(multiplicate) {
    function multiplication(numero) {
        return numero * multiplicate;
    }
    return multiplication;
}
```

Ou assim:

```
function createMultiplicate(multiplicate) {
    return function(numero) { // Sem colocar o nome da função
        return numero * multiplicate;
    }
}
```

Então nós temos uma função que cria já cria um multiplicador e retorna uma outra função. Então se eu quero criar aquelas funções eu faço assim:

```
const duplica = (createMultiplicate(2));
const triplica = (createMultiplicate(3));
const quadriplica = (createMultiplicate(4));
```

Quando eu chamo a função createMultiplicate eu estou passando um parâmetro que é o multiplicador para essa função e daí a outra função interna que usa esse multiplicador então quer dizer que essa função lembra do escopo onde ela está:

```
function createMultiplicate(multiplicate) {
    // multiplicador
    return function(numero) {
        return numero * multiplicate;
    }
}
```

Mas mesmo assim essa função createMultiplicate lembra no momento que eu criei a função const duplica esse multiplicador era 2, então por isso a gente tem **closure** aqui essa função mesmo posteriormente no meu código se eu utilizar ela, ela vai lembrar dos vizinhos dela do multiplicador (multiplicate) que estava no escopo do pai dela. E a função triplica lembra do escopo que na hora da criação dela esse

multiplicador era 3 a quadriplica o multiplicador era 4. A gente chama isso de fechamento de closure, essa é uma função (createMultiplicate) **closure** que está fechando o escopo mesmo depois dele já ter sido usado. Na verdade a gente retornou a função sem executar então por isso a gente vai executar elas aqui:

```
const duplica = (createMultiplicate(2));
const triplica = (createMultiplicate(3));
const quadriplica = (createMultiplicate(4));

console.log(duplica(2));
console.log(triplica(2));
console.log(quadriplica(2));
```

É importante entender isso porque a gente pode retornar qualquer coisa na função gente pode retornar uma outra função, um objeto, os valores primitivos que é o mais comum de se ver enfim a gente pode retornar como a gente viu em um dos exercicíos anteriores no relógio no SetInterval eu retorno aquele SetInterval e uso outra função para parar ou pausar o SetInterval.