

JavaScript

lambda functions

JS

JavaScript

parte 8

aprendendo Javascript

agenda

em JavaScript é possível você passar uma função como parâmetro para outra função, e ela ser armazenada em uma variável. pode parecer estranho em um primeiro momento, mas há patterns avançados que promovem o uso desta funcionalidade.



que nome esquisito é este?

lambda functions

Sendo bem simplista, lambda functions são na verdade funções anônimas, que aprendemos a criar anteriormente. Uma função anônima é basicamente uma função que não tem nome, e normalmente é usada em *callback* functions, eu disse normalmente, não obrigatoriamente.

O nome originou da matemática, o famoso cálculo lambda. Ele trata as funções como valores, fazendo com que elas sejam passadas para outras funções.

Em Javascript é possível inclusive uma função retornar uma função como resultado do seu processamento.

paradigma da

programação funcional

arrow function

() => { }

revendo uma arrow function

estrutura muitas vezes cria dúvidas



```
// arrow function expression - block body
const soma = (foo = 0, bar = 0) => {
  return foo + bar;
}
```

```
// consumindo a função
console.log(soma(6, 1)); // 7
console.log(soma()); // 0
```

quando se usa a notação da arrow function, não há necessidade de se explicitar a palavra reservada `function` o que na prática não parece ser uma boa ideia especialmente para quem está aprendendo a linguagem neste primeiro momento, mas com o tempo você vai se acostumar com esta notação que tem sido a preferida de muitos desenvolvedores.

neste exemplo, a arrow function **soma**, recebe dois parâmetros, inteiros, que por padrão recebem zero, e retorna a soma dos dois. bem simples, mas é para você compreender a sintaxe.

revendo uma arrow function

estrutura muitas vezes cria dúvidas



```
// arrow function expression - concise body
```

```
const duplica = foo => foo + foo;
```

```
const triplica = foo => foo * 3;
```

```
console.log(duplica(2)); // 4
```

```
console.log(triplica(2)); // 6
```

há ainda uma forma mais reduzida de escrever quando a função recebe apenas um parâmetro e retorna uma computação rápida em uma única linha.

eliminamos o uso dos parênteses para os parâmetros e as chaves para o escopo da função.

revendo uma arrow function

estrutura muitas vezes cria dúvidas

```
// arrow function expression - block body
const motd = () => {
  const meses = ['Janeiro', 'Fevereiro', 'Março', 'Abril',
    'Maio', 'Junho', 'Julho', 'Agosto', 'Setembro',
    'Outubro', 'Novembro', 'Dezembro'];
  const today = new Date();
  console.log(`Seja bem-vindo ao dia ${today.getDate()} de
    ${meses[today.getMonth()]} de
    ${today.getFullYear()}.`);
}

// chamada da função
motd();
```

agora ela fica mais confusa quando não recebe nenhum parâmetro e torna-se necessário colocar dois parêntesis para a sintaxe.


no exemplo ao lado, uma função que dá boas-vindas ao dia não precisa receber nenhum parâmetro para sua execução, e nela mesmo está todo o processo de execução.

agora vamos

respirar fundo

lambda function

passando como **parametro**



```
/**
 * arrow function executa que recebe três parâmatros
 * sendo o primeiro uma função e dois outros inteiros.
 * ela retorna a execução da função passada como
 * parâmetro, passado a ela os dois parâmetros inteiros.
 * o resultado é a multiplicação entre os dois.
 */
const executa = (func, foo, bar) => {
  console.log(func(foo, bar)); // 18
}

// chamada da função com uma lambda function no parametro.
executa(function (x, y) { return x * y }, 6, 3);
```

a arrow function executa, receberá três parâmetros, sendo o primeiro uma função, isto mesmo, uma função, e os dois seguintes, dois números inteiros que serão multiplicados entre eles como resposta da função.

neste momento você pode não estar vendo nenhuma vantagem nesta construção, mas quando seu algoritmo se tornar complexo e exigir cálculos rápidos, você vai adorar saber que é possível usar uma construção como esta.

lambda function

passando como **parametro**



```
const executa = (func, foo, bar) => {  
  console.log(func(foo, bar));  
}
```

```
// executa função com lambda function no parametro  
executa(function (x, y) { return x * y }, 6, 3);
```



```
const executa = (func, foo, bar) => {  
  console.log(func(foo, bar));  
}
```

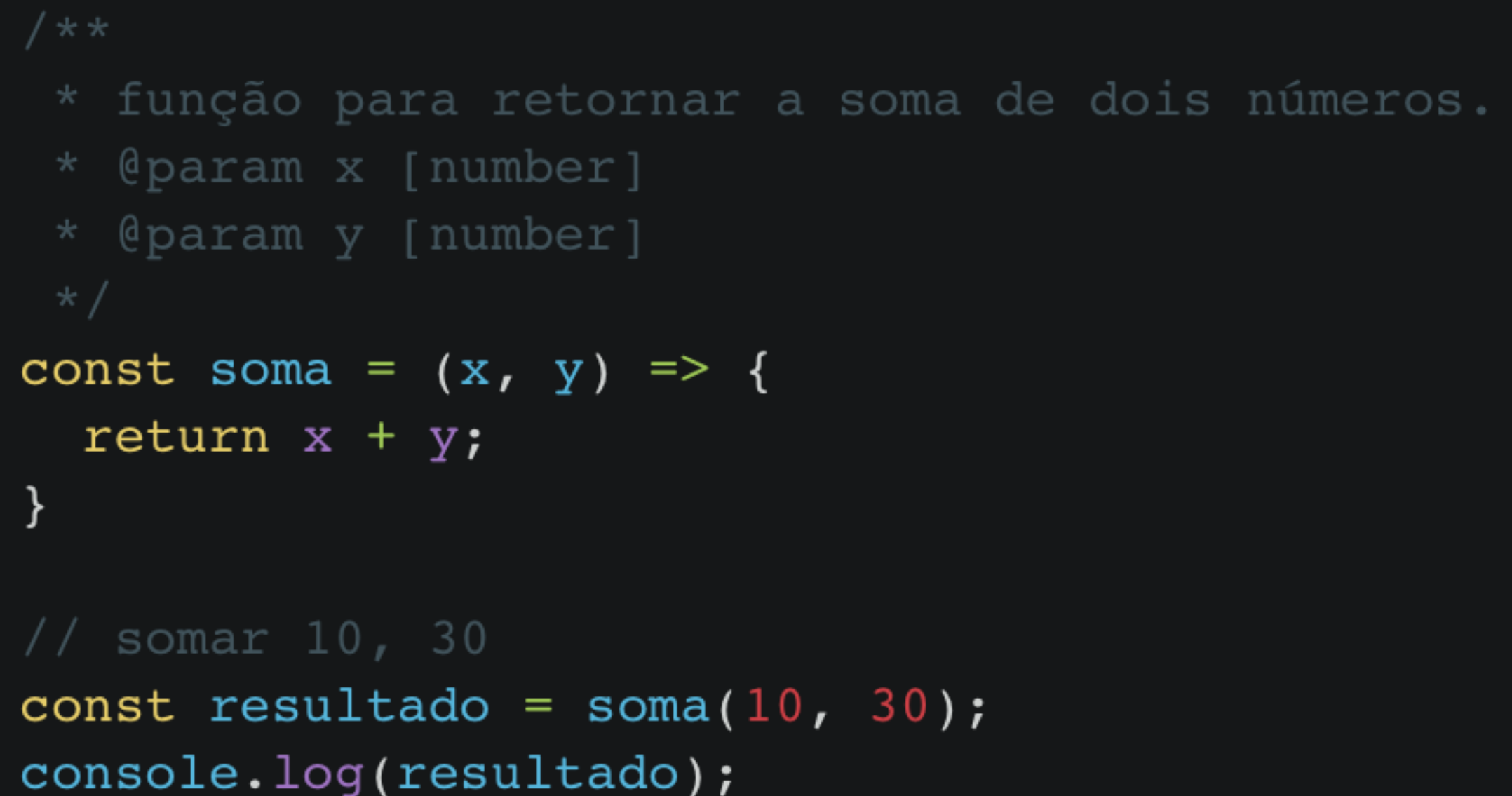
```
// executa função com lambda function no parametro  
executa((x, y) => x * y, 6, 3);
```

...rest parameters

estudo de caso

...rest parameters

você deve construir um programa que some dois números inteiros e positivos, utilizando uma função. Ele deverá somar 10 e 30.



```
/**
 * função para retornar a soma de dois números.
 * @param x [number]
 * @param y [number]
 */
const soma = (x, y) => {
  return x + y;
}

// somar 10, 30
const resultado = soma(10, 30);
console.log(resultado);
```

estudo de caso

...rest parameters

agora você deve construir um programa que retorne a soma de 3 números, usando também uma função. Os números a serem somados são, 10, 30 e 75

```
/**
 * função para retornar a soma de dois números.
 * @param x [number]
 * @param y [number]
 * @param z [number]
 */
const soma = (x, y, z) => {
  return x + y + z;
}

// somar 10, 30 e 75
const resultado = soma(10, 30, 75);
console.log(resultado);
```


estudo de caso

...rest parameters

```
/**
 * função para retornar a soma de dois números.
 * @param x [number]
 * @param y [number]
 * @param z [number]
 */
const soma = (x, y, z) => {
  return x + y + z;
}

// somar 10, 30 e 75
const resultado = soma(10, 30);
console.log(resultado);
```

esta construção, assim como a primeira versão com dois parâmetros tem uma fragilidade. se não passarmos o valor na posição do parâmetro será retornado **NaN**.

estudo de caso

...rest parameters



```
/**
 * função para retornar a soma de dois números.
 * @param x [number]
 * @param y [number]
 * @param z [number]
 */
const soma = (x = 0, y = 0, z = 0) => {
  return x + y + z;
}

// somar 10, 30
const resultado = soma(10, 30);
console.log(resultado);
```

consertar este problema seria bem simples, tornando zero o valor padrão de cada um dos parâmetros.

estamos tornando nosso código mais confiável e robusto, e olha que ainda não estamos usando controle de exceções.

estudo de caso

...rest parameters

agora vamos fazer um programa que some até 50 números inteiros, começando por 10, 20, 45, 76, 87, 14, 55, 81, 66, 03, 20.

quantos parâmetros nós vamos criar na função? 50?



```
/**
 * função para retornar a soma de dois números.
 * @param ...valores (rest parameter)
 */
const soma = (...valores) => {
  let total = 0;
  for (let x = 0; x <= valores.length - 1; x++) {
    total += valores[x];
  }
  return total;
}

console.log(soma(10, 20, 45, 76, 87, 14, 55, 81, 66, 03, 20)); // 477
console.log(soma(10, 20, 45, 76, 87, 14, 55, 81, 66, 03, 20, 20, 30, 40, 50, 77, 12)); // 706
```




```
/**
 * função para retornar a soma de dois números.
 * @param ...valores (rest parameter)
 */
const soma = (...valores) => {
  let total = 0;
  for (let valor of valores) { // itera com cada um dos valores do array de valores.
    total += valor;
  }
  return total;
}

console.log(soma(10, 20, 45, 76, 87, 14, 55, 81, 66, 03, 20)); // 477
console.log(soma(10, 20, 45, 76, 87, 14, 55, 81, 66, 03, 20, 20, 30, 40, 50, 77, 12)); // 706
```



```
/**
 * função para retornar a soma de dois números.
 * @param ...valores (rest parameter)
 */
const soma = (...valores) => valores.reduce((total, unitario) => total + unitario, 0);

console.log(soma(10, 20, 45, 76, 87, 14, 55, 81, 66, 03, 20)); // 477
console.log(soma(10, 20, 45, 76, 87, 14, 55, 81, 66, 03, 20, 20, 30, 40, 50, 77, 12)); // 706
```



```
/**
 * função para retornar a soma de dois números.
 * @param ...valores (rest parameter)
 */
const soma = (...valores) => {
  return valores.reduce((total, unitario) => total + unitario, 0);
}

console.log(soma(10, 20, 45, 76, 87, 14, 55, 81, 66, 03, 20)); // 477
console.log(soma(10, 20, 45, 76, 87, 14, 55, 81, 66, 03, 20, 20, 30, 40, 50, 77, 12)); // 706
```




VAMOS PRATICAR

programming

revendo a prática

princípio básico

1. funções que recebem funções anônimas.
2. funções que retornam funções.
3. as **arrow functions** são bastante interessantes, mas requer uma especial atenção para construir um código que todos possam entender bem.
4. rest parameters que nos permitirá enviar uma quantidade indefinida de parâmetros para uma função.

