ntander Coders 2024.1 | Front-End | #1178

unções e assinaturas

Isaias Soares
Id: 1178008

Funções e assinaturas

O que são funções?

Imagine que todo lugar onde podemos passar um valor, como 10 por exemplo, ou "teste", nós poderíamos passar uma função também. Sendo assim, podemos colocar funções em variáveis, passá-las como parâmetro para outras funções, retornar novas funções de dentro de uma função etc

Em linguagens que tratam funções com menos importância, elas são apenas trechos de código reaproveitáveis. Ou seja, escrevemos algumas linhas de código, damos um nome para elas, e quando usamos esse nome em outro código essas linhas executam. Era muito comum chamar isso de procedimento, rotina, sub-rotina ou outros nomes nas linguagens mais antigas.

Exemplo: Hello World

```
//Cria a função
function helloWorld(){
    console.log('hello world');
}
```

```
//Invoca a função
helloWord();
```

Temos dois momentos no uso das funções, o primeiro quando elas são declaradas (criadas) e o segundo quando elas são invocadas. Uma das diversas vantagens de funções é que, uma vez criadas, elas podem ser usadas quantas vezes quisermos.

Como ilustrado acima, criar uma função é fácil, usamos a palavra `function` seguida do nome da função, seguida de parênteses, e chaves. Dentro das chaves, colocamos os códigos que desejamos rodar quando a função for invocada.

Obs: Nos exemplos abaixo, vamos usar interpolação de strings, no JavaScript essa sintaxe é chamada de "template string", se não conhecer ainda veja na documentação no MDN.

Parâmetros

Parâmetros são utilizados para melhorar a usabilidade da função. São variáveis que declaramos dentro dos parênteses no nome da função. No próximo tópico veremos mais sobre o assunto.

```
function soma(x, y) {
    return x + y
}
```

Retorno

Na maioria dos casos, desejamos obter da função algum valor resultante e utilizá-lo em nosso código. A função de soma acima é útil para fazer apenas uma soma, mas se quisermos utilizá-la no meio de uma conta mais complexa isso não é possível, afinal, tudo o que ela faz é mostrar o resultado na tela.

Podemos modificá-la para que, ao invés de mostrar o resultado na tela, ela devolva ao código que a invocou o resultado. Isso também é muito fácil de fazer, basta no final da função usar o comando `return` e indicar logo após o que você deseja retornar.

Exemplo: Somar com parâmetros e retorno

```
function somar(num1, num2) {
  return num1 + num2;
}

const result = somar(100, 200);
console.log(result); //300
```

Sua função pode fazer o que você quiser, mas só pode retornar um valor! No entanto, nada te impede de retornar um array, um objeto ou uma nova função.

Return como finalizador da função

Sempre que sua função encontra um `return` ela termina imediatamente. Sendo assim, nada após o `return` será executado. Por isso que sempre os colocamos na última linha.

No entanto, você pode usar isso a seu favor para interromper prematuramente uma função. Por exemplo, na nossa soma podemos validar os parâmetros e se não for o que esperamos podemos retornar imediatamente `NaN` que representa "not a number", para isso usaremos a função `isNaN()`.

Exemplo: Somar com parâmetros e retorno se os parâmetros são números

```
function somar(num1, num2) {
  if(isNaN(num1) || isNaN(num2)) return NaN;
  return num1 + num2;
}
```

Ou com ternário:

Exemplo: Somar com parâmetros e retorno se os parâmetros são números (ternário)

```
function somar(num1, num2) {
  return (isNaN(num1) || isNaN(num2)) ? NaN : num1 + num2;
}
```

Funções Anônimas

Se quisermos fazer uma função que vai rodar apenas uma vez ou que vai ser passada como parâmetro para outra função podemos criá-la sem nome. Imagine o seguinte, tenho uma página que tem um botão, assim que a página carregar desejo adicionar um evento ao botão que mostra no console que ele foi clicado.

> Não se preocupe com as funções usadas nesse exemplo. Chegaremos nelas eventualmente e explicaremos direitinho. Por enquanto, foque apenas na função anônima.

Usando um exemplo de uma função que já conhecemos, vamos trabalhar o `find`. Se observarmos, ele recebe uma função por parâmetro, e para cada item do array ele chama essa função passando esse item e a função deve retornar um valor `falsy` ou `truthy`.

```
const nomes = ['Rafael', 'Renan', 'Maicon']
const rafael = nomes.find(function(nome) {
    return nome === 'Rafael'
})
```

Outro exemplo que podemos utilizar, é a função addEventListener que faz isso. Ele espera que passemos por parâmetro a função que desejamos invocar quando o botão for clicado, para isso podemos usar uma função anônima (nossa função deve ter um parâmetro, evt, que representa o evento de clique):

```
const button = document.getElementById('meubotao');
button.addEventListener(function(evt){ console.log('click'); });
```

Podemos colocar todo esse código em uma função anônima que se invoca imediatamente após a declaração também, seria assim:

```
(function(){
    const button = document.getElementById('meubotao');
    button.addEventListener(function(evt){ console.log('click'); });
})();
```

Aqui, colocamos a função toda entre parênteses e um par adicional de parênteses para invocá-la. Isso faz com que ela rode imediatamente e deixe de existir.

Uma forma muito mais comum de usar funções anônimas é colocá-las em variáveis, lembre-se que funções são valores e podem ser armazenadas em variáveis:

```
const somar = function(num1, num2) { return num1 + num2; }
```

Veja, criamos uma função anônima e colocamos dentro da constante somar. Isso produz o mesmo efeito de dar o nome “somar” para a função. Para usá-la não muda nada `somar(1,2)` por exemplo.

Sempre use “const” para garantir que sua função não seja sobrescrita por outra em uma atribuição incorreta. Essa forma de escrever funções costuma ser bem recorrente no JavaScript. Eu mesmo prefiro essa forma do que a padrão, mas gosto ainda mais de arrow functions.

Arrow Functions

Funções anônimas, sejam elas armazenadas em variáveis ou não, são tão comuns em JavaScript que temos uma forma simplificada de criá-las. Se trata de uma nova sintaxe introduzida no ES6 chamada "arrow function" em outras linguagens eles podem ter no nome "lambda" por conta do cálculo lambda que deu origem à programação funcional.

Arrow functions são escritas sem a palavra function, e a quando tem apenas um comando que será retornado as chaves podem ser omitidas, bem como a palavra `return`. Se apenas um parâmetro for usado, os parênteses podem ser omitidos também. Separando os parâmetros do corpo da função teremos uma flecha `=>` por isso o nome arrow function.

Exemplo: Somar com parâmetros e retorno como arrow function

```
const somar = (num1, num2) => num1 + num2;
```

Observe que, por conta de todas as omissões que essa sintaxe nos permite fazer, funções simples são escritas de forma muito prática e reduzida.

No exemplo, o return está implícito e as chaves não são necessárias porque só há um comando. Quando temos mais de um parâmetro, ou nenhum, precisamos manter os parênteses, então nesse não pudemos tirar.

Vamos reescrever o exemplo do botão com arrow function para exemplificar mais:

```
const button = document.getElementById('meubotao');
button.addEventListener(evt => console.log('click'));
```

Nesse caso, omitimos os parênteses, e as chaves, a função ficou super reduzida.

Arrow functions maiores podem ser escritas mantendo as chaves, mas lembre-se: sempre que usar chaves você não poderá mais omitir a palavra `return` para retornar.

```
const teste = () => {
    console.log('1');
    console.log('2');
    console.log('3');
    console.log('testando...');
    return "OK";
}
```

Nesse exemplo a função teste não tem parâmetros, simbolizamos isso usando parênteses vazios antes da flecha. Somos obrigados a manter o `return` porque temos chaves e nela várias expressões, sendo assim, precisamos indicar qual será a que retorna.

Referências e Materiais Complementares

- Arrow functions
- JavaScript Functions

Próximo Tópico