


Prof. Marcos Nava



# Programação de Scripts



# Agenda da Aula

- Atribuição
- Destructuring
- Aritméticos
- Relacionais
- Lógicos
- Unários
- Ternários

# Atribuição

Existem vários operadores que trabalham com atribuição de variáveis.

São eles:

=

+=

-=

\*=

/=

%=

# Atribuição

Exemplo:

```
const a = 7;  
let b = 3;  
b += a;  
console.log(b)  
b -= 4;  
console.log(b)
```

```
b *= 2;  
console.log(b)  
b /= 2;  
console.log(b)  
b %= 2;  
console.log(b)
```

Saída:

```
10  
6  
12  
6  
0
```

# Destructuring

O operador de destructuring tem a função de extrair dados de uma determinada estrutura.

As estruturas podem ser objetos ou vetores.

Para trabalhar com objetos usa-se `{ }` como operador.

Para trabalhar com vetores usa-se `[ ]` como operador.

# Destructuring

Observe o objeto abaixo e responda: Como extrair o nome e a idade ele guardando o valor em outras duas variáveis?

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

# Destructuring

Com destructuring poderíamos fazer assim:

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

Apesar de parecer que criamos duas variáveis e atribuímos o objeto pessoa, na verdade retiramos os atributos do objeto com os nomes "nome" e "idade".

Veja que ficou mais simples o código.

Em uma única linha criamos as variáveis e atribuímos os valores para elas.

Saída:

Ana 5

# Destructuring

Mas, e se eu quisesse ter variáveis com outro nome?

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

Saída:

Ana 5



# Destructuring

O que você acha que acontecerá aqui?

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

Saída:  
undefined true

# Destructuring

E para extrair dados dentro de uma estrutura no objeto?

Veja no objeto que ele possui outro objeto chamado endereço.

Vamos extrair as propriedades existentes logradouro e numero, e uma outra inexistente cep:

```
const { endereco: { logradouro, numero, cep }} = pessoa;  
console.log(logradouro, numero, cep );
```

Saída:  
Rua ABC 1000 undefined

# Destructuring

Cuidado quando usar objetos dentro de objetos.

Se ele não existir você terá problemas:

```
const { conta: { ag, num } } = pessoa;  
console.log(ag, num);
```

Saída:

```
Cannot read properties of undefined (reading 'ag')
```

# Destructuring

Podemos também trabalhar com vetores.

Para exemplo, imagine que você possui um vetor com um único elemento com valor 10 e precisa colocar este valor em uma variável a:

```
const [a] = [10]  
console.log(a);
```

Veja que apesar da sintaxe ser "estranha" ele cria a variável a e joga o primeiro elemento do vetor nela.

10

Saída:

# Destructuring

Vamos a um exemplo mais real:

```
const v = [10, 7, 9, 8];  
const [n1, , n3, , n5, n6 = 0] = v;  
console.log(n1, n3, n5, n6);
```

Temos aqui um vetor chamado de `v` que contém vários elementos.

Em seguida eu faço um destructuring com variáveis posicionais.

Perceba que pulamos alguns itens e tentamos criar outros que não existem.

Saída:  
10 9 undefined 0

# Destructuring

Um exemplo mais complicado...

O que será impresso?

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

Saída:

6

# Destructuring

Outro uso de destructuring é em funções.

Podemos usa-los para já extrair dados de um objeto na passagem de parâmetros:

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

# Destructuring

Vamos agora usar a função:

```
const obj = { max: 50, min: 40 }
```

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

```
console.log(rand({min: 955}));
```

```
console.log(rand({}));
```

Saída:

44

973

460



# Destructuring

E com os vetores?

No exemplo abaixo também estamos usando destructuring para fazer o swap de variáveis se o mínimo for maior que o máximo:

```
function rand([min=0, max=1000])  
{  
  if(min > max) [min, max] = [max, min];  
  const valor = Math.random() * (max-min) + min;  
  return Math.floor(valor);  
}
```

# Destructuring

O que vai imprimir?

```
console.log(rand([50, 40]));  
console.log(rand([992]));  
console.log(rand([,10]));  
console.log(rand([]));
```

Saída:

```
41  
999  
3  
932
```

# Aritméticos

Os operadores aritméticos são os mesmos de outras linguagens:

```
const [a, b, c, d] = [3, 5, 1, 15];  
const soma = a + b + c + d;  
const subtracao = d - b;  
const multiplicacao = a * b;  
const divisao = d / a;  
const modulo = a % 2;  
console.log(soma, subtracao, multiplicacao, divisao, modulo);
```

Saída:  
24 10 15 5 1

# Relacionais

Com relação aos operadores relacionais, JavaScript tem diferenças por ser uma linguagem fracamente tipada.

Geralmente estes operadores comparam conteúdo de variáveis, mas no nosso caso temos como comparar também seus tipos.

Isso ocorre pois um mesmo conteúdo pode estar presente em tipos de variáveis diferentes.

Lembre-se que uma operação relacional sempre dá como resultado verdadeiro ou falso (true or false).

# Relacionais

Comparando se os dados são iguais e estritamente iguais:

```
console.log('1 - ', '1' == 1);  
console.log('2 - ', '1' === 1);  
console.log('3 - ', '3' != 3);  
console.log('4 - ', '3' !== 3);
```

Saída:

```
1 - true  
2 - false  
3 - false  
4 - true
```

# Relacionais

Veja os exemplos abaixo e diga os resultados:

```
console.log('5 - ', 3 < 2);  
console.log('6 - ', 3 > 2);  
console.log('7 - ', 3 <= 2);  
console.log('8 - ', 3 >= 2);
```

Saída:

```
5 - false  
6 - true  
7 - false  
8 - true
```

# Relacionais

Para lidar com objetos temos alguns detalhes interessantes:

```
const d1 = new Date(0);  
const d2 = new Date(0);  
console.log('9 - ', d1 === d2);  
console.log('10 - ', d1 == d2);  
console.log('11 - ', d1.getTime() === d2.getTime());
```

Saída:

```
9 - false  
10 - false  
11 - true
```

# Relacionais

Por último podemos esclarece se undefined é iguar a null:

```
console.log('12 - ', undefined == null);  
console.log('13 - ', undefined === null);
```

Saída:

12 - true

13 - false



# Lógicos

Os operadores lógicos seguem as outras linguagens e são:

&& - E

|| - Or

# Unários

Alguns operadores unários já são usados naturalmente em programação, ! e -:

```
let v1 = true;
```

```
let n1 = 1;
```

```
console.log('v1', v1);
```

```
console.log('v1', !v1);
```

```
console.log('n1', n1);
```

```
console.log('n1', -n1);
```

Saída:

v1 true

v1 false

n1 1

n1 -1

# Unários

Os outros operadores unários são ++ e --:

```
n1++;  
console.log('n1', n1);  
++n1;  
console.log('n1', n1);  
--n1;  
console.log('n1', n1);  
n1--;  
console.log('n1', n1);
```

Saída:

```
n1 2  
n1 3  
n1 2  
n1 1
```

# Unários

Eles podem ser pré-fixados ou pós-fixados, mas muda um pouco o comportamento dentro de fórmulas:

```
console.log('n1', n1++);  
console.log('n1', ++n1);  
console.log('n1', n1--);  
console.log('n1', --n1);
```

Saída:

```
n1 1  
n1 3  
n1 3  
n1 1
```

# Ternários

O operador ternário tem este nome porque ele possui três partes:

`<condicao> ? <verdadeiro> : <falso>`

Primeiro é avaliado a condição lógica e depois, conforme o resultado seja verdadeiro ou falso, temos um resultado:

```
const resultado = nota => nota >= 6 ? 'Aprovado' : 'Reprovado';  
console.log(resultado(5.9));  
console.log(resultado(6.5));  
console.log(resultado(3));  
console.log(resultado(10));
```

Saída:  
Reprovado  
Aprovado  
Reprovado  
Aprovado