Javascript – Módulo 4 – B7Web

Sumário

Aula 01 – Métodos de String (1/3)

.length
.indexOf()

Aula 02 – Métodos de String (2/3)

slice() →

sintaxe: nome.slice(10,15)
→ O primeiro parâmetro indica a posição do início da seleção.
→ O segundo parâmetro indica a posição final da seleção.
Neste exemplo, selecionamos todos os itens entre a posição 10 e 15.

Obs. 1: Assim como em um array, a primeira posição é a de número zero.
Obs. 2: Se colocarmos apenas 1 parâmetro, ele começa a seleção daquela posição, seleciona todas as outras posições até o final.
Obs. 3: Se colocarmos como parâmetro um número negativo, ele seleciona a partir do final, a quantidade de ítens que for definido. ex.: nome.slice(-4) -> seleciona os quatro últimos ítens.
Obs. 4: Também podemos contar a seleção a partir do final, porém indicarmos também até onde termina a seleção, sem a obrigatoriedade de selecionar todos até o final. Ex.: nome.slice(-4,-2) -> este comando selecionaria, a partir do item -4 até o item -2. Neste exemplo, deixando o último item de fora.

---

substring() → O substring tem as mesmas funcionalidades do slice, porém ele não trabalha contando as posições a partir do final, ou seja, com números negativos como parâmetros.

---

substr() → Comando parecido com o substring. A diferença aqui está no segundo parâmetro, que ao invés de indicar a posição do último ítem da seleção, ele indica a quantidade de itens que serão selecionados. Ex.: nome.substr(10,15) -> Este comando selecionaria, a partir do item de posição 10, ele mesmo e os próximos 14 itens.

Obs. 1: aqui, o substr aceita números negativos no primeiro parâmetro para indicar a seleção de trás para a frente, sendo que o segundo parâmetro precisa ser um número positivo.

Obs. Geral: Dentre estes 3 métodos, o substr() geralmente é o mais utilizado.

Aula 03 – Métodos de String (3/3)

replace()
replaceAll()
let resultado = nome.replace('carros','motos')

---

toUpperCase()

toLowerCase()
let resultado = nome.toUpperCase();

---

concat() → adiciona os parâmetros indicados na string, sendo possível adicionar quantos parâmetros forem desejados.
Obs.: Este comando não altera a string original. Ex.: nome.concat(' Teixeira', ' Quadros')
a princípio, este comando gera o mesmo resultado de uma concatenação utilizando o sinal de '+'. Ex.: nome + ' Teixeira' + ' Quadros'.
let resultado = nome.concat(' Teixeira', ' Quadros')

---

trim() → Este comando remove todos os espaços da string que aparecerem antes de qualquer caractere e todos os espaços que aparecem após o último caractere. Ex. 1: ' Daniel ' -> nome.trim() -> 'Daniel'. Ex. 2: ' Daniel Teixeira ' nome.trim() -> 'Daniel Teixeira'
let resultado = nome.trim()

---

charAt() → retorna o caractere que está na posição indicada.
Obs.: nome.chartAt(3) tem o mesmo resultado de nome.substr(3, 1) ou ainda nome[3].
let resultado = nome.charAt(3)

---

split() → Significa dividir
Adicionamos como parâmetro o caractere(símbolo ou espaço) que queremos identificar como divisor. O comando retorna um array com cada parte desta divisão em uma posição distinta. Ex.: nome.split(' ') -> ['Daniel', 'Teixeira', 'Quadros']
Outro exemplo muito funcional para o comando split é conseguirmos colocar uma string com números separados por vírgulas em um array. Ex.: nome = '1,2,3,4,5' -> nome.split(',') -> ['1', '2', '3', '4', '5'].

Outros Métodos Strings – Extras (Fontes externas ao B7W)

toLocaleString('pt-BR', {style: 'currency', currency: 'BRL'}) – tranforma para o padrão moeda, neste exemplo para o Real do Brasil, ex:
n1.toLocaleString('pt-BR', {style: 'currency', currency: 'BRL'})
outros exemplos:
Dólar - n1.toLocaleString('pt-BR', {style: 'currency', currency: 'USD'})
Euro - n1.toLocaleString('pt-BR', {style: 'currency', currency: 'EUR'})
(Fonte: Curso em Vídeo)

shift() → remove o primeiro elemento de um array e retorna esse elemento. Este método muda o tamanho do array.
https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array/shift

[7,8,9,6,3]
num.shift() → este comando retorna o número 7.
Resultado → [8,9,6,3]

O método **unshift()** adiciona um ou mais elementos no início de um array e retorna o número de elementos (propriedade length) atualizado.

https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array/unshift

[8.9.6.3]
num.unshift(7) → retorna o número 5, que é a quantidades de itens que o array ficará(num.length)
Resultado → [7,8,9,6,3]

[7,8,9,6,3]
num.unshift(4,2) → retorna o número 7, que é a quantidades de itens que o array ficará(num.length)
Resultado → [4,2,7,8,9,6,3]

O método **pop()** remove o último elemento de um array e retorna aquele elemento.
https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array/pop

[8,9,6,3]
num.pop() → este comando retorna o número 3
Resultado → [8,9,6]

splice() → altera o conteúdo de uma lista, adicionando novos elementos enquanto remove elementos antigos.
https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array/splice

para substituirmos o valor 9 da posição 1, por 5, podemos fazer da seguinte maneira. Obs, o comando retornará com o valor substituído. Neste exemplo, o número 9.

[8,9,6,3]
num.splice(1,1,5) → Onde o primeiro 1 indica o índice que será substituído, o segundo 1 indica quantos índices serão alterados a contar do primeiro valor alterado. E o número 5 neste exemplo, é o valor que irá substituir o número 9.
Resultado → [8,5,6,3]. Ao executarmos este comando, este será o novo valor contido no nosso array num, mas o resultado retornado desta operação será o número 9, ou seja o valor que foi retirado do array.

Desta forma, utilizando este mesmo comando, podemos substituir, de uma só vez, diversos valores ao mesmo tempo. Exemplo:
num.splice(0,3,9,6,7) → Com este único comando, estamos informando que: Queremos alterar a partir do índice **0**, ele mesmo e os próximos 2 índices (por isso o número **3**, ou seja, ele + 2) e queremos colocar os valores **9**, **6** e **7**. Cada um em uma das posições da sequencia.
Resultado → array num fica assim: [9,6,7,3].
Valor retornado pela execução do comando: [8,5,6], ou seja os valores que foram substituídos.

OBS: Se no primeiro campo a ser declarado no comando splice(), utilizarmos números negativos, o comando irá contar a posição dos índices de trás para frente, ou seja, se utilizarmos o valor -1 na primeira posição, ele substituirá o último item deste array. Exemplo:
[9,6,7,3]
num.splice(-1,1,5) → O comando retornará o número 3 que é o número substituído e o array ficará assim: [9,6,7,5]

num.splice(-2,2,3,8) → O comando começará as substituições a partir da penúltima posição(**-2**), na quantidade de 2 substituições (**2**) e substituirá os valores existentes pelos valores (**3,8**). O comando ainda retornará os valores substituídos, ou seja, os valores (7,5), desta maneira [7,5] e deixará o nosso array assim: [9,6,3,8]

Outro exemplo curioso. Se utilizarmos o índice -1 para iniciarmos da última posição, porém utilizarmos um número maior que 1 para indicarmos a quantidade de posições a serem substituídas, o comando irá substituir a última posição pelo valor indicado e irá criar novas posições para adicionar os valores restantes. Exemplo:
[9,6,3,8]

num.splice(-1,3,7,2,5) → O comando irá começar as substituições a partir da posição 3(última posição, indicado pelo valor **-1**) e criará mais duas posições (última + 2 → indicado pelo número **3**). Neste caso, retornando apenas o valor 8, que foi o único valor substituído e deixando o nosso array num da seguinte maneira:
[9,6,3,7,2,5]

Aula04 – Métodos de Numbers

Aula 05 – Métodos array (1/4)

to.String() → Podemos tranformar array em string

join() → Tranforma um array em uma string separando os itens com o parâmetro definido
let res = lista.join('-')

```
let lista = ['Ovo','Farinha','Corante','Massa']
let res = lista.join('-')
let res = lista
console.log(res)
Ovo-Farinha-Corante-Massa
```

Aula 06 – Métodos array (2/4)

delete → Detela o valor contido na posição indicada, porém, não deleta a posição em que este item estava.

```
delete lista[1]
['Ovo', …, 'Corante', 'Massa']
```

concat → Este comando concatena dois arrays em um só.

```
let lista = ['Ovo','Farinha','Corante','Massa']
let lista2 = ['prato','talher','forno']
let res = lista.concat(lista2)
['Ovo', 'Farinha', 'Corante', 'Massa', 'prato', 'talher', 'forno']
```

Aula 07 – Métodos array (3/4)

map → Este comando mapeia um array e é utilizado para salvar os itens deste array em outro array. Utilizamos este comando para fazer algum tipo de modificação desejada. Utilizamos um exemplo para multiplicar os valores do array 'lista' por 2 e salvar no array 'lista2'. Observe a estrutura no exemplo abaixo.

```
let lista = [45, 4, 9, 16, 25]
```

```
let lista2 = []
lista2 = lista.map(function(item){
  return item*2
})
```

Este exemplo tem a mesma funcionalidade do 'for' abaixo. porém, em alguns casos, é mais interessante utilizar o 'map'. Para reaproveitar uma 'function' é um caso.

```
for(let i in lista){
  lista2.push(lista[i] * 2)
}
```

filter → O comando filter também verifica todas as informações de um array, porém ele retorna apenas aqueles que satisfizerem uma condição determinada (true or false), salvando este resultado em outro array.

```
lista2 = lista.filter(function(item){
  if(item < 20){
    return true
  }else{
    return false
  }
})
```

Neste exemplo acima, filtramos todos os números menores que 20.

every → O comando 'every', similar ao filter, também retorna true or false, porém, ele só retorna true se todos os itens do array satisfizerem a condição determinada, caso contrário, se pelo menos 1 item não satisfizer a esta condição, ele retornará false.

```
lista2 = lista.every(function(item){
  if(item>20){
    return true
  }else{
    return false
  }
})
```

some → Este comando também retorna true or false, porém, diferentemente do 'every', ele retornará true se pelo menos um dos elementos satisfizerem a condição determinada e só retornará false se nenhum item satisfizer esta condição.

```
lista2 = lista.some(function(item){
  if(item>20){
    return true
  }else{
    return false
  }
}
```

Podemos, neste exemplos de true or false, utilizar a verificação ternária, para resumirmos o código:

```
(item > 20)? true : false
```

Aula 08 – Métodos array (4/4)

find → Procura alguma informação no array → Retona o item, o index e também o array inteiro. nesta ordem respectivamente. Se colocarmos apenas um parâmetro, ele retornará apenas o item, se colocarmos apenas dois parâmetros, ele retornará o item e o seu respectivo index.

```
lista2 = lista.find(function(item, index, array){
  return (item == 16)? true: false
})
```

Veja um exemplo mais prático:

```
let lista = [
  {id:1, nome: 'Daniel', sobrenome: 'Quadros'},
  {id:2, nome: 'Lisi', sobrenome: 'Cruz'},
  {id:3, nome: 'Gabi', sobrenome: 'Camargo'}
]
let pessoa = lista.find(function(item){
  return (item.sobrenome == 'Cruz')? true : false
})
{id: 2, nome: 'Lisi', sobrenome: 'Cruz'}
```

findIndex - retorna aposição do item solicitado.

```
lista2 = lista.findIndex(function(item, index, array){
  return (item == 16)? true: false
})
```

Aula 09 – Datas (1/3)

Objeto **Date** → Podemos trabalhar com todas informações com relação a tempo

```
let d = new Date()→
```

Ele vem com a data atual e este comando pega esta informação do dispositivo em que for executado:

```
console.log(d)
Thu May 06 2021 16:04:21 GMT-0300 (Brasilia Standard Time)
```

toDateString() → forma mais resumida.

```
console.log(d.toDateString())
Thu May 06 2021
```

d.toUTCString() - retorna a hora do GMT

```
console.log(d.toUTCString())
Thu, 06 May 2021 19:06:43 GMT
```

A classe 'Date' permite ser adicionado a ela, **até 7 parâmetros:**
let d = new Date(ano, mês, dia, hora, minuto, segundos, milésimos)

```
let d = new Date(2021, 4, 6, 16, 16, 30, 777)
```

```
console.log(d.toString())
Thu May 06 2021 16:16:30 GMT-0300 (Brasilia Standard Time)
```

Obs.: importantíssima → a declaração de mês, começa com o número 0, ou seja, janeiro=0... dezembro=11. De acordo com o Buniek, é algo sem sentido, mas é assim que funciona, ou seja se acostume.

Ao invés de colocarmos todos estes parâmetro, podemos declarar apenas 1 parâmetro (string) para obter o mesmo resultado do comando acima

```
let d = new Date('2021-04-06 16:21:30:777')
console.log(d.toString())
Tue Apr 06 2021 16:21:30 GMT-0300 (Brasilia Standard Time)
```

Observe o padrão de declaração: separação de ano, mês e dia com '**-**', depois disso, um espaço e a separação entre hora, minuto, segundo e mile segundo com '**:**'.

Nos dois exemplos acima, precisamos declarar no mínimo o ano e o mês

```
let d = new Date('2021-04')
console.log(d.toString())
Wed Mar 31 2021 21:00:00 GMT-0300 (Brasilia Standard Time)
```

Curiosidade importante sobre datas. Quando definimos o parâmetro de tempo com o número **0**:
→ **com o toString**, o número 0 aponta para 31/12/1969 21:00:00 GTM (Horário Padrão de Brasília)
→ **com o toUTCString**, o número 0 aponta para 01/01/1970 00:00:00 GTM

```
let d = new Date(0)
console.log(d.toString())
Wed Dec 31 1969 21:00:00 GMT-0300 (Brasilia Standard Time)

console.log(d.toUTCString())
Thu, 01 Jan 1970 00:00:00 GMT
```

Isto acontece, porque o Javascript começa a contar os mile segundos a partir de 1970. Mas não somente o Javascript, mas outras linguagens como o PHP também utilizam este padrão.
Isto não impede que utilizemos datas anteriores a isto.

Podemos declarar e calcular uma data utilizando **apenas os mile segundos**, por exemplo. O Javascript calculará o tempo a partir de 1970. Para calcularmos o tempo anterior a este ano, utilizamos valores negativos. ex:

```
let d = new Date(123156463)
console.log(d.toUTCString())
Fri, 02 Jan 1970 10:12:36 GMT

let d = new Date(-123156463)
console.log(d.toUTCString())
Tue, 30 Dec 1969 13:47:23 GMT
```

Raramente será utilizado esta informação, mas é importante saber como funciona.

Aula 10 – Datas (2/3)

getFullYear() → retorna o ano atual em um número de 4 dígitos.
```
let d = new Date()
let res = d.getFullYear()
```

getMonth() → retorna o mês atual em número. Lembrando que no Javascript os meses começam com o número 0.
```
let res = d.getMonth()
```

getDay() → Retorna o dia da semana atual em número. Obs.:no Javascript os dias da semana são contados de **0 a 6**, sendo que o 0 é o domingo.
```
let res = d.getDay()
```

getDate() → retorna o dia do mês atual em número.
```
let res = d.getDate()
```

getHours() → retorna a hora atual em números.
```
let res = d.getHours()
```

getMinutes() → retorna o minuto atual em números.
```
let res = d.getMinutes()
```

getSeconds → retorna o segundo atual em números.
```
let res = d.getSeconds()
```

getMilliseconds() → retorna o mile segundo atual em números (3 dígitos).
```
let res = d.getMilliseconds()
```

getTime() → Retona o **time stamp**, ou seja, o tempo contado mile segundos atual desde o dia 01/01/1970 às 00:00:00. em um único número. Exemplo:
```
let res = d.getTime()
console.log(res)
1620335264888
```

Date.now() → Retorna o momento atual com o time stamp.
```
let res = Date.now()
```

Aula 11 – Datas (3/3)

Parâmetro **set** → configurar, alterar.

setFullYear() → Este comando altera o ano, também pode ser utilizado para alterar o mês e o dia.

```
let d = new Date()
d.setFullYear(2022)
let res = d
console.log(res)
Fri May 06 2022 18:21:06 GMT-0300 (Brasilia Standard Time)
```

setMonth() → Este comando altera o mê. Lembrando que em Javascript os meses são contados de 0 a 11.
```
d.setMonth(11)
```

```
let res = d
console.log(res)
Mon Dec 06 2021 18:23:39 GMT-0300 (Brasilia Standard Time)
```

E assim por diante, da mesma forma que pegamos as informações de tempo com o '**get**', aqui, conseguimos alterar essas informações com o '**set**'.: **setDay(), setDate(), setHours(), setMinutes(), setSegonds() e setMilliseconds()**.

Podemos também alterar todas as unidades de tempo utilizando os comandos relativos ao tempo desejado. Podemos somar e subtrair:

```
d.setDate(d.getDate() + 5)
let res = d
console.log(res)
Tue May 11 2021 18:32:39 GMT-0300 (Brasilia Standard Time)

d.setHours(d.getHours() + 5)
let res = d
console.log(res)
Thu May 06 2021 23:37:34 GMT-0300 (Brasilia Standard Time)
```

E assim por diante.

Aula 12 – Matemática

Math -Método que calcula fórmulas matemáticas.

Math.PI -Retorna o valor de PI
```
let val = Math.PI
```

Três funções básicas para arredondamento de números:

Math.round → Arredonda o número para o **inteiro mais próximo**(Obs. **o .5 arredonda para cima**, exemplo: 3.5 arredonda para 4).
```
let val = Math.round(3.5)
```

Math.floor → floor(chão)Arredonda o número para o **inteiro inferior.**
```
let val = Math.floor(3.9)
```

Math.ceil → ceil(teto) Arredonda para o **inteiro superior.**
```
let val = Math.ceil(3.1)
```

Math.abs → Retorna o valor absoluto, ou seja, ignora se o número é negativo.
```
let val = Math.abs(-3.1)
```

Math.min() -Retorna o menor valor de uma sequência.
```
let val = Math.min(7, 100, 600, 20, 3, -9)
```

Math.max() → Retorna o maior valor de uma sequência.

Math.random() → Retorna um número aleatório entre 0 e 1.
```
let val =Math.random()
```

Obs. Na maioria das vezes em que precisamos pegar um número aleatório, não queremos um número entre 0 e 1. Para isto utilizamos a seguinte técnica:
- Utilizamos o Math.random e multiplicamos pelo número máximo que queremos, neste exemplo utilizaremos o 100.
- Pegamos o resultado e arredondamos ele para baixo para garantir que ele nunca passará do nosso número máximo.

```
let val = Math.floor(Math.random() * 100)
```

Obs.: Neste exemplo, ele nunca será 100, para isto precisaríamos multiplicar por 101.

Outros métodos:

Seno, cosceno e tangente: Math.sin , Math.cos e Math.tan

Ainda existem muitos outros, porém são mais específicos e são facilmente encontrados no Google.

Aula 13 – Intervalos(1/2) Timers

setInterval → função que define um intervalo entre execuções, este intervalo é definido em mile segundos. Possui dois parâmetros: No primeiro declaramos o que será executado e no segundo parâmetro, definimos de quanto em quanto tempo(em mile segundos) será executado.
`setInterval(showTime, 1000)`

clearInterval → Função que zera os parâmetro da função setInterval, fazendo com que o contador configurado pare.
`clearInterval(timer)`

JAVASCRIPT:

```
let timer
function comecar(){
    timer = setInterval(showTime, 1000)
}
function parar(){
    clearInterval(timer)
}
function showTime(){
    let d = new Date()
    let h = d.getHours()
    let m = d.getMinutes()
    let s = d.getSeconds()
    let txt = `${h}:${m}:${s}`
    document.querySelector('.demonst').innerHTML = txt
}
```

HTML:

```
<body>
    <h1>Bem vindo</h1>
    <button onclick="comecar()">Começar</button>
    <button onclick="parar()">Parar</button>
    <div class="demonst"></div>
```

NAVEGADOR:

Aula 14 – Intervalos(2/2) Timers

setTimeout → Recebe dois parâmetros. O primeiro é o que ele irá executar, neste nosso exemplo, executar uma função. O segundo parâmetro é utilizado para definir em quanto tempo ele executará esta função. Este tempo é definido em mile segundos.

```
function comecar(){
    setTimeout(function(){
        document.querySelector('.demonst').innerHTML = "Rodou!!!"
    }, 2000)
}
```

Neste segundo exemplo, armazenamos o setTimeout na variável timer, para poder trabalharmos com ela. Aqui, utilizamos esta variável na função 'parar' com o comando clearTimeout.

ClearTimeout → Comando que zera os parâmetros do comando setTimeout, tem o objetivo de parar o timer antes que a função do setTimeout seja executada.

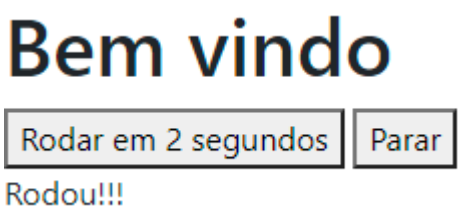
`clearTimeout(timer)`

```
let timer
function comecar(){
    timer = setTimeout(function(){
        document.querySelector('.demonst').innerHTML = "Rodou!!!"
    }, 2000)
}
function parar(){
    clearTimeout(timer)
}
```

HTML:

```
<h1>Bem vindo</h1>
<button onclick="comecar()">Rodar em 2 segundos</button>
<button onclick="parar()">Parar</button>
<div class="demonst"></div>
```

NAVEGADOR:



Aula 15 – Template String (`let texto = `número ${n}``)

ECMA SCRIPT 6 (ES6+)

```
//Template String
let nome = 'Daniel'
let idade = 90
//let frase = 'Meu nome é '+nome+' e eu tenho '+idade+' anos e ano que vem eu farei'+(idade + 1)+' anos.'
let frase = `Meu nome é ${nome} e eu tenho ${idade} anos e ano que vem eu farei ${idade+1}`
console.log(frase)
```

Aula 16 – Descontruindo Objeto(1/2) - ECMA SCRIPT 6 (ES6+)

Exemplo de objeto para os exercícios:

```
let pessoa = {
    nome: 'Daniel',
    sobrenome: 'Quadros',
    idade: 90,
    social:{
        facebook: 'fdtq',
        instagram: 'idtq'
    },
    nomeCompleto: function(){
        return `${this.nome} ${this.sobrenome}`
    }
}
```

Chamando um item deste objeto:

```
console.log(pessoa.nome)
```

Chamando um item do objeto que está dentro do objeto pessoa:

```
console.log(pessoa.social.facebook)
```

Chamando uma função que está dentro do objeto pessoa:

```
console.log(pessoa.nomeCompleto())
```

MANEIRA ANTIGA DE DESCONSTRUÇÃO:

```
let nome = pessoa.nome
let idade = pessoa.idade
let sobrenome = pessoa.sobrenome
console.log(nome, sobrenome, idade)
```

MÉTODO DE DESCONSTRUÇÃO DE OBJETO:

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

MÉTODO DE DESCONSTRUÇÃO DE OBJETO ALTERANDO O NOME DA VARIÁVEL DE SAÍDA DE CADA ITEM:

```
let { nome:pessoaNome, sobrenome, idade } = pessoa;
console.log(pessoaNome, sobrenome, idade)
```

MÉTODO DE DESCONSTRUÇÃO DE OBJETO COM DEFINIÇÃO DE UM VALOR PADRÃO:
Neste exemplo, retiramos o item 'idade do objeto para exemplificarmos.

```
let pessoa = {
    nome: 'Daniel',
    sobrenome: 'Quadros',
    social:{
        facebook: 'fdtq',
        instagram: 'idtq'
    },
    nomeCompleto: function(){
        return `${this.nome} ${this.sobrenome}`
    }
}
```

Aqui, para adicionarmos um
item que não existe no objeto da desconstrução, neste exemplo, o item idade. Adicionamos o item que precisamos, mais o sinal de igual(=) e definimos o valor a ser recebido.

```
let { nome, sobrenome, idade = 90} = pessoa;
console.log(nome, sobrenome, idade)
```

Obs. **Se o objeto tiver o item com um valor adicionado**, a desconstrução ignora o valor padrão definido na desconstrução e assume o valor do item que ven do objeto. Ex. se no objeto tivesse o item idade com o valor 88, a desconstrução assumiria o valor 88 e não 90 para idade.

Aula 17 – Descontruindo Objeto(2/2) - ECMA SCRIPT 6 (ES6+)

Utilizado o seguinte objeto para os dois exemplos abaixo:

```
let pessoa = {
    nome: 'Daniel',
    sobrenome: 'Quadros',
    idade: 90,
    social:{
        facebook: 'fdtq',
        instagram: 'idtq'
    },
    nomeCompleto: function(){
        return `${this.nome} ${this.sobrenome}`
    }
}
```

COMO DESCONSTRUIR O OBJETO QUE ESTÁ DENTRO DE UM OBJETO:

(facebook e instagram), itens do objeto que está dentro do objeto

```
let{facebook, instagram} = pessoa.social
console.log(facebook,instagram)
```

Pegando itens do objeto e itens internos do objeto interno:

```
let{nome, idade, social:{instagram}} = pessoa
console.log(nome,idade, instagram)
```

Para os próximos exemplos, utilizado o objeto abaixo:

```
let pessoa = {
    nome: 'Daniel',
    sobrenome: 'Quadros',
    idade: 90,
    social:{
        facebook: 'fdtq',
        instagram: {
            url: '@daniel',
            seguidores: 1000
        }
    }
}
```

Pegando um item do objeto interno do objeto interno:

```
let{nome, idade, social:{instagram:{url}}} = pessoa
console.log(nome,idade, url)
```

Também podemos pegar todos os elementos deste objeto interno:

```
let{nome, idade, social:{instagram:{url, seguidores}}} = pessoa
console.log(nome,idade, url, seguidores)
```

Neste próximo exemplo, pegamos a variável url, porém a renomeamos para instagram

```
let{nome, idade, social:{instagram:{url:instagram}}} = pessoa
console.log(nome,idade, instagram)
```

Podemos também pegar o objeto interno por completo:

```
let{nome, idade, social:{instagram}} = pessoa
console.log(nome,idade, instagram)
```

Utilizando desconstrução de objeto em uma função:

Exemplo 1, sem a desconstrução de objeto:

```
function pegarNomeCompleto(obj){
    return `${obj.nome} ${obj.sobrenome}`
}
console.log(pegarNomeCompleto(pessoa))
```

Exemplo 2, com a desconstrução de objeto. Neste caso, o objeto já é desconstruído nos parâmetros da função:

```
function pegarNomeCompleto({nome, sobrenome}){
    return `${nome} ${sobrenome}`
}
console.log(pegarNomeCompleto(pessoa))
```

Aula 18 – Desconstruindo Arrays - ECMA SCRIPT 6 (ES6+)

Para todos os exemplos desta aula utilizamos o seguinte array:

```
let info = ['Daniel Quadros', 'Daniel', 'Quadros', '@daniel']
```

Lembrete: A grande diferença entre um array e um objeto, é que o array não tem um nome como chave(index), o array tem um index(chave) que é um número que vai de 0 ao infinito.

Entre os colchetes, definimos o nome que cada variável que receberá um item do array terá. e após o igual, identificamos o array da desconstrução.

```
let[nomeCompleto, nome, sobrenome, instagram] = info
console.log(nomeCompleto, nome, sobrenome, instagram)
```

Esta forma capta os itens do array, na ordem em que estão posicionados dentro deste array.

Se declararmos apenas uma variável na desconstrução, por exemplo. Será capturado apenas o elemento da posição 0 do array.

```
let [a] = info
console.log(a)
```

Conclusão: Com base nesta informação, verificamos que a desconstrução de um array precisa ser feita na ordem em que os itens internos deste array estão organizados, pois será nesta ordem que obteremos a saída para armazenarmos nas variáveis da desconstrução.

Formas de utilizar apenas algumas informações deste array mas que não estão em uma sequência:

```
let [nomeCompleto, a, b, instagram] = info
console.log(nomeCompleto, instagram)
```

Neste exemplo acima, capturamos todas as informações deste array, porém, utilizamos apenas a primeira e a última variável criada. Desta forma, criamos duas variáveis inúteis.

Como fazer esse tipo de captura sem a criação de variáveis inúteis:

```
let [nomeCompleto, , , instagram] = info
console.log(nomeCompleto, instagram)
```

Na declaração de desconstrução, deixamos vazio o espaço do item que não desejamos utilizar. Desta forma não criamos variáveis inúteis (não há desperdício de memória).

Neste exemplo abaixo, criamos um array e já fizemos a desconstrução dele. Obs.: Não é muito utilizado, mas é importante entendermos o código caso nos deparemos com esse formato.

```
let [nome, sobrenome] = ['Daniel', 'Quadros']
console.log(nome, sobrenome)
```

Pode ser útil, no caso em que, quando estivermos construindo em sistema e tivermos que declarar diversas variáveis inicias. desta forma, podemos criar todas elas, já desconstruídas de uma única vez.

Assim, como nos objetos, podemos definir um valor padrão para um item que não tenha sido declarado:

```
let [nome, sobrenome, idade=90] = ['Daniel', 'Quadros']
console.log(nome, sobrenome, idade)
```

Existe também a possibilidade de criarmos um array e fazermos a desconstrução dele através de uma função:

```
function criar(){
    return [1,2,3]
}
let numeros = criar()
let[a,b,c] = numeros
console.log(a,b,c)
```

Outra forma de utilizarmos função e economizar variáveis:

```
function criar(){
    return [1,2,3]
}
let[a,b,c] = criar()
console.log(a,b,c)
```

Obs.: Este exemplo também funciona com desconstrução de objetos.

Aula 19 – Arrow Functions - ECMA SCRIPT 6 (ES6+)

Arrow Function → Também conhecida como função anônima.
Arrow (Flecha) Function (Função)

Há três formas de declararmos funções:

1ª, forma mais antiga(tradicional):

```
function somar(x, y){
    return x + y
}
console.log(somar(5,7))
```

2ª, forma que já estava disponível desde o ECMA Script 5

```
let somar = function(x, y){
    return x + y
}
console.log(somar(5,7))
```

Acima, adicionamos uma função dentro de uma variável, e por este motivo, a função não recebe nome, pois ela é chamada através da variável em que está contida.

3ª forma. É parecida com a segunda, porém ainda mais resumida:

Obs.: Podemos criar em uma variável qualquer (var, let ou const). De acordo com o Bonieky, geralmente é utilizado const, porém neste exemplo, ele optou por usar let, apenas para manter o padrão dos exemplos anteriores.

```
let somar = (x,y)=>{
    return x + y
}
console.log(somar(5,7))
```

Declaramos a variável e adicionamos ao recebimento dela, o(s) parâmetro(s). E para indicar que esta declaração é uma função utilizamos os operadores '**=>**' juntos, formando uma flecha mesmo.

Obs.: Após a flecha, **se a função tiver apenas uma ação específica**, temos a opção de abrirmos as chaves '{}' ou não.

Se não quisermos utilizar as chaves'{}' (Lembrando que este modelo funciona apenas em função que tenham apenas uma ação específica)no início da declaração dos comandos da função, também podemos declarar as ações que ela realizará diretamente após a flecha. Sem a necessidade do comando 'return'.

```
let somar = (x,y)=> x+y
console.log(somar(5,7))
```

Existe uma outra opção, mas que pode ser utilizada quando a função recebe apenas 1 parâmetro:

1º Exemplo, utilizando o **1º padrão de arrow function** que vimos:

```
let letrasNome = (nome) => {
    return nome.length
}
console.log(letrasNome('Daniel'))
```

2º Exemplo, utilizando o **2º padrão de arrow function** que vimos:

```
let letrasNome = (nome) => nome.length
console.log(letrasNome('Daniel'))
```

Lembrando que, aqui, na declaração da função, não precisamos utilizar as chaves '{}', porque esta função tem apenas uma ação.

3º Exemplo. **Esta é a forma que podemos utilizar nas funções que tem apenas 1 parâmetro:**

```
let letrasNome = nome => nome.length
console.log(letrasNome)
```

Qual é a diferença? Quando temos apenas 1 parâmetro, **os parênteses '()'** que definem o campo para a declaração dos parâmetros **são opcionais**.

Obs.: A única **diferença entre** a declaração de uma **arrow function e os outros** métodos de declaração de funções, é que na arrow function **não há o objeto 'this'.**

Aula 20 – Operador Spread (...)

Operador Spread:

Para adicionar informações de um array dentro de outro array sem utilizar operador Spread, precisaríamos utilizar o comando push para cada item do array que queremos inserir. Com o Operador spread fica mais fácil. Veja o exemplo:

Esta forma adicionamos o array ‘numeros’ no início do array ‘outros’:

```
let numeros = [1,2,3,4]
let outros = [...numeros,5,6,7,8]
console.log(outros)
(8) [1, 2, 3, 4, 5, 6, 7, 8]
```

Sem o operador spread o resultado não seria o esperado, veja o exemplo:

```
let numeros = [1,2,3,4]
let outros = [numeros,5,6,7,8]
console.log(outros)
(5) [Array(4), 5, 6, 7, 8]
```

Podemos utilizar o **operador spread em array e principalmente em objetos**, que normalmente é onde mais é utilizado.

```
let info = {
    nome: 'Daniel',
    sobrenome: 'Quadros',
    idade: 40
}
let novaInfo = {
    ...info,
    cidade: "Gov. Celso Ramos",
    estado: 'Santa Cataina',
    pais: 'Brasil'
}
console.log(novaInfo)
{nome: 'Daniel', sobrenome: 'Quadros', idade: 40, cidade: 'Gov. Celso Ramos', estado: 'Santa Cataina', …}
```

Podemos criar uma função para reaproveitamento do operador spread:

```
function adicionarInfo(info){
    let novaInfo = {
        ...info,
        status: 0,
        dataCadastro:'.....'
    }
    return novaInfo
}
console.log(adicionarInfo({nome:'Daniel', sobrenome:'Quadros'}))
{nome: 'Daniel', sobrenome: 'Quadros', status: 0, dataCadastro: '.....'}
```

Aula 21 – Operador Rest (...)

Operador rest (...) é muito menos utilizado do que o operador spread, mas é importante conhecer

Para entendermos, vamos verificar o primeiro exemplo, sem o operador rest:

```
function adicionar(numeros){
    console.log(numeros)
}
adicionar(5)
```

Neste exemplo, criamos uma função que tem a capacidade de receber apenas 1 parâmetro.

Vamos ver a mesma função, porém agora adicionando o operador rest nela:

```
function adicionar(...numeros){
    console.log(numeros)
}
adicionar(5, 7, 9, 11)
```

Agora, com o operador rest, a função está preparada para receber 1 ou mais parâmetros indefinidamente.

Exercício utilizando os operadores spread e rest o rest juntos:

```
function adicionar(nomes, ...novosNomes){ //operador rest
    let todos = [...nomes,...novosNomes] //operador spread
    return(todos)
}
let nomes = ['Daniel','Lisi']
let outros = adicionar(nomes, 'Gabi','Julia')
console.log(outros)
(4) ['Daniel', 'Lisi', 'Gabi', 'Julia']
```

Aula 22 – Includes e Repeat – ECMA SCRIPT 6 (ES6+)

Includes & Repeat

includes → serve para strings e arrays.
Utilizamos ele para verificar se há um elemento dentro de uma variável. Ele retorna true ou false.
Veja o exemplo 1:

```
let lista = ['ovo', 'café','arroz','feijão','macarrão']
console.log(lista.includes('carne'))
false
```

Exemplo 2:

```
let nome = 'Daniel'
console.log(nome.includes('b'))
false
```

---

repeat → Esta função, como o próprio nome já diz, gera uma repetição na quantidade de vezes que for definida. Exemplo 1:

```
console.log('x'.repeat(20))
```

Exemplo 2:

```
let nome = 'Daniel '
console.log(nome.repeat(20))
```

Aula 23 – Objeto: Key, Values e Entries – ECMA SCRIPT 6 (ES6+)

Funciona tanto em objeto como em array

Vajamos:

```
let lista = [1,2,3,4]
console.log(typeof lista)
object
```

```
let lista = {nome: 'Daniel', idade: 40}
console.log(typeof lista)
object
```

Um conceito interessante é que **tanto array, como objects são do tipo object**

Por isto, para os comando **'keys', 'values' e 'entries' utilizamos o comando 'Object'** antecessor. Veja os exemplos:

Utilizado o array abaixo para os próximos exemplos

```
let lista = ['ovo','macarrão','feijão','pipoca']
```

keys → Retorna os índices do array.:

```
console.log(Object.keys(lista))
(4) ['0', '1', '2', '3']
```

---

values → Retorna os valores do array:

```
console.log(Object.values(lista))
(4) ['ovo', 'macarrão', 'feijão', 'pipoca']
```

---

entries → Cria um array de duas posições para cada item do nosso array original. Contendo na primeira posição(0) o index do item e na segunda posição(1) o próprio item.

```
console.log(Object.entries(lista))
(4) [Array(2), Array(2), Array(2), Array(2)]
```

```
Array(4)
▸ 0: (2) ["0", "ovo"]
▸ 1: (2) ["1", "macarrão"]
▸ 2: (2) ["2", "feijão"]
▸ 3: (2) ["3", "pipoca"]
```

Vamos ver em objetos:
Para os exemplos abaixo, utilizaremos o seguinte objeto:

```
let pessoa = {
    nome: 'Daniel',
    sobrenome: 'Quadros',
    idade: 40
}
```

---

keys:

```
console.log(Object.keys(pessoa))
(3) ['nome', 'sobrenome', 'idade']
```

---

values:

```
console.log(Object.values(pessoa))
(3) ['Daniel', 'Quadros', 40]
```

---

entries:

```
console.log(Object.entries(pessoa))
(3) [Array(2), Array(2), Array(2)]
```

```
▼Array(3) 1
  ▶0: (2) ["nome", "Daniel"]
  ▶1: (2) ["sobrenome", "Quadros"]
  ▶2: (2) ["idade", 40]
```

Aula 24 – String, padStart, padEnd – ECMA SCRIPT 6 (ES6+)

padEnd → Este comando define o número de caracteres que uma string deve ter, se por acaso, não for adicionado o número de caracteres exigidos, podemos definir um determinado caractere para preencher os valores que não foram adicionados. Veja o exemplo:

```
let telefone = '5'
console.log(telefone.padEnd(9, '*'))
5********
```

No exemplo acima, utilizamos como parâmetros do padEnd o número **9** que definiu a quantidade de caracteres mínimos aceitáveis e no segundo parâmetro, adicionamos o **'*'** que foi utilizado para preencher o que não havia valor.

PadStart:

```
console.log(telefone.padStart(9, '*'))
********5
```

padStart → Já, o padStart, ao contrário do padEnd, ele preenche os valores faltantes, antes dos valores declarados.

Exemplo prático:

```
let cartao = '1234567890123456'
let lastDigits = cartao.slice(-4)
console.log(lastDigits.padStart(16, 'X'))
XXXXXXXXXXXX3456
```