Coleções JavaScript

O que são Coleções?

- Conjunto de Dados agrupados.
- Temos Arrays, Maps e Sets.
- propriedade Length e podem ser indexados

Arrays

É uma sequencia de Dados **modificável, Não tipada, indexada** e possui duas formas de ser criada: de forma literal e com construtor

```
const Array1 = [1,2,true,"pedro",5]; // Array LIteral
const Array2 = new Array(1,2,true,"pedro",5); // Array Object
console.log(Array1) // [1,2,true,"pedro",5]
console.log(Array2) // [1,2,true,"pedro",5]
```

Indexação de Arrays

Para acessar os elementos de um Array, podemos usar um **indexador**(popularmente chamado de índice)

```
const array1 = [1,2,3,4,5]
console.log(array1[1]) // 2
```

Da mesma forma que também podemos modificar os valores.

```
const array1 = [1,2,3,4,5]
array1[1] = 9
console.log(array1) // [1,9,3,4,5]
```

?

```
const array1 = [1,2,3,4,5]

console.log(array1[8])
array1[10] = 4
console.log(array1)
```

A classe Array define metodos **não estáticos** para todos os Objetos do **tipo Array**, que podemos usar para **Modificar** o estado do Array.

- Entre eles temos:
- **Concat,**
- **every,**
- **fill,**
- **filter,**
- **find,**
- **ForEach,**
- **includes,**
- **Map,**
- **Pop,**
- **Push, reduce, reverse, Shift, Sort, Slice, Some, Splice...**

Alguns termos importantes

- **Mutação** (*Mutation*):

Mutação é conhecido como a **mudança de estado** de um objeto. Alguns métodos mutam o Array ao invés de trazer uma cópia nova.

- **Cópia superficial** e **Cópia profunda** (*Shallow Copy*, *Deep Copy*):
  - Referência ao Array original.
  - Referência própia.

Array.Push & Pop

- **Push** Serve Para **Adicionar** um Item ou vários itens ao **Final do Array**. Este metodo retorna o **Length** do Array modificado

```
const array1 = [1,2,3,4,5]
console.log(array1) // [1,2,3,4,5]
const length = array1.push('value')
console.log(array1) // [1,2,3,4,5,'value']
console.log(length) // 6
```

- **Pop**, por outro Lado, serve para **remover** um elemento do **final do Array** . este método retorna o **elemento removido.**

```
const array1 = [1,2,3,4,5]
const item_removido = array1.pop();
console.log(array1) // [1,2,3,4]
console.log(item_removido) // 5
```

Array.Shift & Unshift

O método **Shift** é outra versão do Pop, só que ele **remove** o **primeiro elemento** e retorna o **elemento removido.**

```
const Array1 = [1,2,3,4,5]
const primeiro_elemento = Array1.shift()
console.log(Array1) // [2,3,4,5]
console.log(primeiro_elemento) // 1
```

o método **Unshift** é uma versão do push, só que ele **adiciona** o elemento no **começo do Array** e retorna o **length** do array modificado.

```
const Array1 = [1,2,3,4,5]
const length = Array1.unshift('adição')
console.log(Array1) // ['adição',1,2,3,4,5]
console.log(length) // 6
```

Array.Splice

Os métodos vistos anteriormente apenas nos deixam mudar o começo e o fim de um Array, mas v uma posição escolhida? Para isso usamos o **Splice**, que tem a função de **remover**/**adicionar** elementos em um dado **indice.**

temos como sintaxe: **Array.Splice**(**Start,Count, Item1, Item2... ItemN**)

```
const array1 = [1,2,3,4,5]
// começando no indice 1, remova 0 elementos
array1.splice(1,0)
console.log(array1) // [1,2,3,4,5]
// Começando no indice 2, remova 1 elemento e adicione banana
array1.splice(2,1,"Banana")
console.log(array1) // [1,2,'banana',4,5]
// Começando no indice 2, remova 1 elemento e adicione morango e cereja
array1.splice(2,1,'morango','cereja')
console.log(array1) // [1,2,'morango','cereja',4,5]
```

Array.Slice & Fill

O Método **Slice** serve para dividir um array em um **subArray** dado um **indice** no começo e no fim.

```
const array1 = [1,2,3,4,5]
// divida o array no indice 1 até o 4 (Atenção: O 4 não é incluido)
const subArray = array1.slice(1,4)
console.log(subArray) // [2,3,4]
```

O método **Fill** serve para preencher posições do Array com um **valor Fixo.** aceita como parametros o valor, o indice de começo e fim.

```
const array1 = [1,2,3,4,5]
array1.fill(9)
console.log(array1) // [9,9,9,9,9]
// Preenche com o número 4 do indice 2 até o indice 4 (o indice 4 não está incluido)
array1.fill(4,2,4)
console.log(array1) // [9,9,4,4,9]
```

Array.Sort & Reverse

O Método **Sort** Serve para ordenar um Array com base em uma **função fornecida** Que aceita 2 parâmetros: **currentValue e NextValue.** se nenhuma função for fornecida, **Sort** ordenará o Array em ordem baseado nos **Valores Unicode**.

O método **Reverse** serve para, como diz o nome, reverter os **elementos e posições** de um array. este método retorna uma **Copia superficial** do array modificado.

```
const array1 = [1,2,3,4,5]
const reference = array1.reverse()
array1[0] = 20
console.log(array1) // [20,4,3,2,1]
console.log(reference) // [20,4,3,2,1]
```

Função Testadora

Uma **Função Testadora** é aquela que faz um teste e **retorna Verdadeiro** se o teste foi passado, ou **retorna falso** se o teste não for passado.

```
function quemEMaior (param1, param2){
    if(param1 > param2){
        return true
    }
    else{
        return false
    }
}
function quemEMaior2 (param1, param2){
    return param1 > param2
}
const quemEMaior3 = (param1, param2) => param1 > param2
```

Array.Includes & Find

O método **Includes** verifica se um valor passado como parâmetro **está no array**. Retorna **verdadeiro** caso esteja, ou **falso** caso contrário.

```
const frutas = ['maçã','banana','morango']
console.log(frutas.includes('banana')) //true
console.log(frutas.includes('banan')) //false
```

Já o método **Find** permite procurar o **primeiro valor** que passe a **função testadora** colocada como parâmetro. retorna o

```
const array1 = [1,2,3,4,5,6]
const PrimeiroPar = array1.find((element) => element % 2 === 0)
console.log(PrimeiroPar) // 2
```

Manipulação De Arrays

para manipular e percorrer um Array, podemos usar um **For Convencional**

```
const array1 = [1,2,3,4,5]
for(let i = 0; i < array1.length; i++){
    console.log(array1[i])
    // 1
    // 2
    // 3
    // 4
    // 5
}
```

Mas há métodos da Classe Array específicos que são mais rápidos que um for convencional, e que podemos usar.

Array.ForEach

Este método nos permite percorrer um Array de forma mais dinâmica. A sintaxe dele pode ser descrita como:

```
array1.forEach(CallBack)
```

Em que **CallBack** é uma função que pode receber 3 parâmetros, sendo os últimos 2 opcionais: **Element, index e Array.**

Element: O elemento no Array que está sendo percorrido
index: o index do elemento percorrido.
array: o array que está sendo usado.

Este Padrão de função (**Element,index,array**) é usado em quase todos os métodos de mapeamento de Arrays, **como Map, reduce, filter, Foreach, Every, Some, Find entre outros.**

```
const array1 = [1,2,3,4,5]

const CallBack = function(element,index){
    console.log(`na posição ${index} tem o elemento ${element}`)
}
array1.forEach(CallBack)
// ou ----------------------
array1.forEach(function(element,index){
    console.log(`na posição ${index} tem o elemento ${element}`)
})
// ou ----------------------
array1.forEach((element) =>{
    console.log(element)
})
```

Array.Map

Este método É similar ao ForEach, mas ele **retorna um Array** que podemos modificar baseado no Array percorrido. usa o mesmo tipo de **função** de 3 parâmetros.

```
const array1 = [1,2,3,4,5]

const MapArray = array1.map((Element) =>{
    return Element * 2
})
// ou ---------------
const MapArray2 = array1.map(Element => Element * 2)
console.log(MapArray) // [2,4,6,8,10]
console.log(MapArray2) // [2,4,6,8,10]
```

Array.Filter

este método **filter** serve para **filtrar** elementos de um array, baseados em uma **função Testadora** passada como parâmetro. Está função recebe os mesmos 3 parâmetros e retorna um **Array Filtrado** baseado na **condição da função testadora**.

```
const array1 = [1,2,3,4,5,6,7,8,9,10]
const todosImpares = array1.filter((element) => element % 2 !== 0)
console.log(todosImpares) // [1,3,5,7,9]
```

Array.Every & some

- O método **Every** percorre o Array e retorna verdadeiro se **todos os elementos** passam pela **função testadora** .
- Já o método **Some** funciona de forma igual ao Every, mas ele só precisa que **um elemento** no array passe a **função testadora** para retornar verdadeiro.

```
const array = [1,2,3,4,7,8,'agua']

const allNumbers = array.every((element) => typeof element === 'number')

const someNumber = array.some((element) => typeof element === 'number')

console.log(allNumbers) // false
console.log(someNumber) // true
```

Objetos do tipo ArrayLike

Alguns Objetos do javascript Podem parecer Arrays, só que na verdade, não são. Estes objetos, como Arrays, **possuem a propriedade Length e podem ser indexados,** mas não são herdados da classe array e portanto não possuem os métodos ForEach, Filter, etc...

Mas porque isso é importante?

Alguns métodos do **DOM** comuns não retornam Arrays, mas sim **objetos ArrayLike**.
uns exemplos de estes objetos ArrayLike são **HTMLCollection e NodeList**.

```
const Paragrafos = document.getElementsByClassName('paragrafos')

Paragrafos.forEach((element) =>{
    console.log(element.innerHTML)
})
//Uncaught typeError
// Oops, HTMLCollection não possue um método ForEach
```

Array.From

Como **Objetos ArrayLike** não possuem propriedades de um Array é necessario converter eles para um objeto do **Tipo Array**, usando o método estático **From**. Este metodo aceita como parametro um objeto iterador ou um objeto ArrayLike, e o transforma em Array.

```
const Paragrafos = document.getElementsByClassName('paragrafos')
const Arrayparagrafos = Array.from(Paragrafos)
Arrayparagrafos.forEach((element) => console.log(element.innerHTML))
```

Sets

Sets são conjuntos de dados que **não se repetem**. é uma sequência de dados única e é declarado usando o construtor da Classe Set. O construtor set recebe como parâmetro um **Array** ou um **objeto Iterador**.

```
const set1 = new Set(1,2,3,4,5) //errado!
console.log(set1) // Erro
const set2 = new Set([1,2,3,4,5]) //Certo!
console.log(set2) //{1,2,3,4,5}
const set3 = new Set([1,2,3,3,4]) // ?
console.log(set3) // ?
```

Os métodos da classe Set são usados principalmente para comparar dois sets.

- **Intersection**
- **Difference**
- **symmetricDifference**
- **isSubsetOf**
- **isSupersetOf**
- **Has**
- **size**
- **union**
- **isDisjointFrom**