# JS-12: JavaScript Moderno

*Aula 2: Funções, “this”, Arrays, Web Storage, Parâmetros Rest e Spread Operator*

## Funções e Arrow Functions

A versão 6 do ECMAScript trouxe diversas novidades. Uma das mais conhecidas é uma nova opção para criação/declaração de funções na linguagem denominada *Arrow Function*. Além de uma sintaxe mais otimizada (menos verbosa) as *Arrow Functions* também possuem um comportamento diferente quanto ao escopo de sua execução interferindo também no comando **this**, bastante utilizado em diversos cenários.

No entanto, vamos discutir primeiramente as duas formas que tínhamos até então para declarar funções no JavaScript: usando as *function declarations* e as *function expressions*.

**Function Declarations**
As declarações de funções (function declaration) é a forma mais comum de criarmos funções em JavaScript. Veja sua sintaxe:

```
function nomeDaFuncao(parametros) {
    // o que a função faz
}
```

- A principal vantagem da sintaxe de declaração é que podemos chamar/executar uma função antes mesmo de de a declararmos na ordem do código. Isso ocorre por conta de um comportamento do JavaScript chamado *HOISTING*. Esse comportamento faz com que o JavaScript ponha automaticamente todas as declarações no topo do código em tempo de execução.

  Ex:

  ```
  somar(15, 5)
  function somar(valor1, valor2) {
      return valor1 + valor2
  }
  ```

  O código acima não vai dar problema pois embora a execução da função esteja antes de sua declaração, por conta do comportamento de *HOISTING* do JavaScript, em tempo de execução as declarações de funções são trazidas para “cima”, ou seja, antes

das chamadas delas e é por isso que funciona. Para ilustrar esse comportamento, é como se o código acima ficasse estruturado assim em tempo de execução:

```
function somar(valor1, valor2) {
        return valor1 + valor2
}
somar(15, 5)
```

**Function Expressions**

As funções no JavaScript também são valores (assim como strings, boleanos e números). Isso quer dizer que podemos guardar/declarar uma função numa variável. Esse tipo de declaração chamamos de function expressions. Veja:

```
let nomeDaFuncao = function(parametros) {
        // o que a função faz
}
```

- Ao contrário das declarações de funções, as function expressions não sofrem HOISTING. Desse modo, não podemos executar uma função (criada com function expression) antes de sua criação pois teremos um erro na execução.

  Ex:

  ```
  somar(20, 8) // ERRO: somar is not a function
  let somar = function(valor1, valor2) {
          return valor1 + valor2
  }
  ```

  O código acima dará erro pois estamos tentando executar a função antes de a criarmos. Como criamos a função somar usando uma function expression, OBRIGATORIAMENTE só podemos utilizá-la depois de sua declaração e NUNCA antes.

**Arrow Functions**

As *Arrow Functions (funções flecha)* talvez sejam as queridinhas dos desenvolvedores JavaScript atualmente. Elas permitem a escrita de funções de uma maneira menos verbosa e são muito, mas muito utilizadas mesmo hoje em dia.

Elas seguem as mesmas regras das **function expressions** inclusive sua forma de execução (só podem ser utilizadas depois de sua criação, NUNCA antes).

```
let nomeDaFuncao = (parametros) => {
    // o que a função faz
}
```

Uma vantagem clara das *Arrow Functions* é a flexibilidade da sua declaração. Apesar da sintaxe acima mostrada ser a sintaxe padrão, dependendo do caso, a forma de declaração desse tipo de função pode ser ainda mais reduzida. Vejamos abaixo:

- Função sem parâmetros e várias linhas de código:

```
let nomeDaFuncao = () => {
    // o que a função faz
}
```

- Função com MAIS DE UM PARÂMETRO e com várias linhas de código:

```
let nomeDaFuncao = (parametro1, parametro2,...) => {
    // o que a função faz
}
```

- Função com APENAS UM PARÂMETRO e com várias linhas de código:

```
let nomeDaFuncao = parametro => {
    // o que a função faz
}
```

- Função com APENAS UM PARÂMETRO e com UMA LINHA DE CÓDIGO APENAS:

```
let nomeDaFuncao = parametro => // o que a função faz
```

*Obs.: Quando falarmos de funções de Array (High Order Functions) a utilidade das Arrow Functions ficará ainda mais acentuada assim como sua sintaxe simplificada deixa o código muito mais limpo e direto de ser compreendido.*

## Entendendo o comando “this”

O comando “this” no JavaScript se comporta de forma diferente dependendo de onde e como ele é executado. De forma geral, **this** se refere ao objeto atual do contexto de sua execução. Pode ocorrer alterações também em seu comportamento se seu código está sendo executado em modo estrito (strict mode = “use strict”) ou não.

Vamos entender melhor!

- **O “this” por padrão se refere ao objeto global,** isso quer dizer que se executarmos um *console.log(this)* numa tag script ou num arquivo *.js* diretamente (fora de funções e objetos), o console mostrará nos navegador por exemplo o objeto “Window”.

- O **“this” dentro de uma função** também se refere ao **objeto global no modo “não estrito”**; no **modo estrito (“use strict”)** retorna **undefined**.

- O **“this” dentro de um método de um objeto** se refere ao **próprio objeto do contexto** da função.
  Ex:

```
const obj1 = {
        mostrarIdade: function() {
                console.log(this) // this = obj1
        }
}
```

- **Arrow Functions NÃO** guardam o escopo do **“this”**, desse modo, o **“this”** dentro de função desse tipo **vai buscar o objeto mais alto na hierarquia que guarde um contexto (geralmente uma função)**. Se não encontrar, retorna o **objeto global**.
  Ex:

```
const obj2 = {
        mostrarIdade: () => {
                console.log(this) // this = window (objeto global no navegador)
        }
}
```

- Podemos alterar o contexto do **“this”** para qualquer função usando o comando **bind()**. A única exceção são as **Arrow Functions**. O **“this”** de uma função desse tipo é SEMPRE aquele do contexto onde ela foi CRIADA, desse modo, não podemos redefinir o **“this”** das *Arrow Functions*.

## Arrays e High-Order Functions

Talvez um dos tipos de dados que é mais utilizado nas linguagens de programação é o tipo Array, também conhecido como Matriz ou Vetor. Trabalhar com lista de dados é uma constante no dia a dia da programação e saber as funções que nos permitem manipular esses conjuntos de dados é não apenas crucial como também bastante produtivo, já que pode nos salvar diversas linhas de código que podem ser completamente desnecessárias.

Vamos conhecer um pouco os métodos e propriedades que o ECMAScript nos disponibiliza para lidar com esse tipo na linguagem.

1. **pop()**: remove o último item do array e o retorna. Também altera a lista original não exibindo mais o último item.
   Ex:
   ```
   const animais = [“Gato”, “Cachorro”, “Papagaio”]
   let ultimoItem = animais.pop()
   console.log(ultimoItem) // “Papagaio”
   console.log(animais) // [“Gato”, “Cachorro”]
   ```

2. **shift()**: remove o primeiro item do array e o retorna. Também altera a lista original não exibindo mais esse primeiro item.
   Ex:
   ```
   const animais = [“Gato”, “Cachorro”, “Papagaio”]
   let primeiroItem = animais.shift()
   console.log(primeiroItem) // “Gato”
   console.log(animais) // [“Cachorro”, “Papagaio”]
   ```

3. **push(item):** adiciona um item no final do array. Retorna a quantidade de itens na lista atualizada.
   Ex:
   ```
   const animais = [“Gato”, “Cachorro”, “Papagaio”]
   let quantidade = animais.push(“Rato”)
   console.log(quantidade) // 4
   console.log(animais) // [“Gato”, “Cachorro”, “Papagaio”, “Rato”]
   ```

4. **unshift(item):** adiciona um item no início do array. Retorna a quantidade de itens na lista atualizada.
   Ex:
   ```
   const animais = [“Gato”, “Cachorro”, “Papagaio”]
   let quantidade = animais.unshift(“Rato”)
   console.log(quantidade) // 4
   console.log(animais) // [“Rato”, “Gato”, “Cachorro”, “Papagaio”]
   ```

5. **join(separador):** define como os itens de uma lista serão exibidos com o uso de um separador.
   Ex:
   ```
   const animais = [“Gato”, “Cachorro”, “Papagaio”]
   console.log(animais.join(“ | “)) // Gato | Cachorro | Papagaio
   ```

6. **length:** propriedade que retorna a quantidade de itens num array.
   Ex:

   **const** animais = [“Gato”, “Cachorro”, “Papagaio”]
   console.log(animais.**length**) *// 3*

7. **splice(posicao, [quantidade]):** permite a exclusão de itens de um determinado array. O parâmetro *quantidade* não é obrigatório, porém, se não especificado, todos os itens que vierem depois da *posição* informada serão excluídos da lista também. Se especificado *quantidade* como “1”, apenas o item da *posição* informada será excluído. Retorna um array contendo o item ou os itens excluído(s).
   Ex:

   **const** animais = [“Gato”, “Cachorro”, “Papagaio”]
   **let** excluido = animais.**splice(0, 1)**
   console.log(excluído) *// [“Gato”]*
   console.log(animais) *// [“Cachorro”, “Papagaio”]*

8. **reverse():** faz com que a ordem dos itens no array fique ao contrário.
   Ex:

   **const** animais = [“Gato”, “Cachorro”, “Papagaio”]
   animais.**reverse()**
   console.log(animais) *// [“Papagaio”, “Cachorro”, “Gato”]*

**High-Order Functions**

Antes de mais nada, o que seriam *High-Order Functions*? Bem… a melhor definição para esses tipos de função é:

> *High-Order Functions são funções que podem receber outras funções como argumento ou retornar funções como valor/resultado de uma operação.*

Passar funções como parâmetro de outra é extremamente comum não apenas em linguagens como o JavaScript como também em outras como o PHP. Quando discutirmos **CALLBACKS** e **PROMISES** isso ficará ainda mais evidente, porém para onde estamos iremos discutir as *High-Order Functions* aplicadas aos Arrays.

Vejamos algumas delas:

1. **forEach():** essa função se comporta como uma espécie de looping especializado, percorrendo os itens do array e colocando-os como parâmetro de uma função que passamos como argumento para esse método.
   Veja:

   **const** nomes = [“Antônio”, “Gabriela”, “José”]
   nomes.**forEach(function(item, index) {**

```
        console.log(item) // mostra cada nome no console
})
```

Como podemos perceber, o forEach recebe como parâmetro uma função (chamada de função de “callback”) que será executada na passagem por cada item do array. Essa função recebe vários argumentos, porém os mais usados são os 2 primeiros. O primeiro argumento representa o item atual do array e o segundo argumento retorna o índice do item atual. Apenas o primeiro parâmetro é obrigatório. Os demais são todos opcionais.

Lembre-se que também podemos usar Arrow Functions no lugar de funções anônimas como no exemplo anterior. Veja como o código ficaria:

```
const nomes = [“Antônio”, “Gabriela”, “José”]
nomes.forEach((item, index) => {
        console.log(item)
})
```

**OU AINDA...**

```
const nomes = [“Antônio”, “Gabriela”, “José”]
nomes.forEach(item => console.log(item))
```

2. **map():** essa função mapeia (transforma) um array de um tipo em outro. Na realidade, esse método utiliza as informações de um array e retorna um novo array com os dados necessários.
   Ex:

```
const usuarios = [
        {user: “usuario1”, idade: 32, ativo: true},
        {user: “usuario2”, idade: 27, ativo: true},
        {user: “usuario3”, idade: 21, ativo: false},
]
const idades = usuarios.map(usuario => {
        return usuario.idade
})
console.log(idades) // exibe: [32,27,21]
```

   Pega o array de usuários e mapeia esse array para um array apenas com as idades dos usuários. Ou seja, de um array com várias informações geramos um novo array (totalmente transformado) com as idades dos usuários e exibimos essa lista.

3. **filter():** filtra os itens de um array com base na condição fornecida e retorna um array com os itens que atendem essa condição.
   Ex:

```
const usuarios = [
        {user: “usuario1”, idade: 32, ativo: true},
        {user: “usuario2”, idade: 27, ativo: true},
        {user: “usuario3”, idade: 21, ativo: false},
]
const menos30 = usuarios.filter(usuario => {
        return usuario.idade < 30
})
/*
        Exibe usuários com menos de 30 anos: [
          {user: “usuario2”, idade: 27, ativo: true},
         {user: “usuario3”, idade: 21, ativo: false}
       ]
*/
console.log(menos30)
```

4. **find()** e **findIndex():** procura o primeiro item que atender a condição fornecida. *find()* retorna o item encontrado ou undefined (se não encontrado). *findIndex()* retorna o índice do item encontrado ou -1 (se não encontrado).
   Ex:

```
const usuarios = [
        {user: “usuario1”, idade: 32, ativo: true},
        {user: “usuario2”, idade: 27, ativo: true},
        {user: “usuario3”, idade: 21, ativo: false},
]
const usuarioEncontrado = usuarios.find(usuario => {
        return usuario.user === “usuario2”
})
const indiceUsuario = usuarios.findIndex(usuario => {
        return usuario.user === “usuario2”
})
/*
        Exibe no console...
        find: {user: “usuario2”, idade: 27, ativo: true}
        findIndex: 1
*/
console.log(“find:”, usuarioEncontrado)
console.log(“findIndex:”, indiceUsuario)
```

5. **reduce(função, valor_inicial_acumulador):** reduz as informações processadas de um array a um valor único.
   Ex:

```
const somaIdades = usuarios.reduce((acumulador, usuario) => {
```

```
        return acumulador + usuario.idade
}, 0)
/*
        Exibe no console...
        Soma das Idades: 80
*/
console.log("Soma das Idades:", somaldades)
```

A função *reduce()* recebe 2 valores: a função de callback e o valor inicial do acumulador (neste caso, zero). Ao passar por cada item presente no array "*usuarios"*, pegamos o valor do acumulador e somamos com a idade do usuário. Esse valor é retornado e associado ao acumulador que ao final é retornado contendo a soma de todas as idades presentes nos usuários da lista.

6. **sort():** ordena os itens de um array. Por padrão, tenta ordenar de forma alfabética, porém também aceita uma função que defina como os itens deverão ser ordenados. Essa função sempre deve retornar os valores *0, 1 ou -1 (ou, via de regra, qualquer número positivo ou negativo)*. -1 (ou um número negativo) indica que o *item1* deve vir antes do *item2*; 0 indica que o *item1* e o *item2* são iguais, logo tanto faz quem vem antes ou depois; 1 (ou num número positivo) indica que *item1* deve vir depois de *item2*. Ex:

```
const animais = ["Gato", "Arara", "Baleia"]
animais.sort()
console.log(animais) // ["Arara", "Baleia", "Gato"]

const numeros = [12, 15, 10, 2, 3]
numeros.sort((valor1, valor2) => valor1 - valor2)
console.log(numeros) // [2,3,10,12,15]

const palavrasComAcento = ["Feijão", "Melão", "Arroz", "Água"]
palavrasComAcento.sort((palavra1, palavra2) => {
        return palavra1.localeCompare(palavra2)
})
console.log(palavrasComAcento) // ["Água", "Arroz", "Feijão", "Melão"]
```

Para compararmos palavras acentuadas e ordená-las da forma correta, precisamos a função especial **localeCompare()** que compara strings e as posiciona na ordem correta.

## Web Storage: trabalhando com sessionStorage e localStorage

A Web Storage API é um recurso presente nos navegadores que possibilita a armazenagem de pequenas quantidades de dados locais (dados não sensíveis), ou seja, no front-end.

Tanto a *sessionStorage* quanto a *localStorage* fornecem os mesmos métodos para manipulação dessas informações, porém as duas se diferenciam na forma como esses dados são mantidos.

Vejamos:

- **sessionStorage:** armazena uma informação no browser de forma parcialmente permanente, isso porque o dado só estará disponível enquanto a aba do navegador que criou aquela informação estiver aberta. Assim que a aba ou a janela for fechada, a informação se perderá.

- **localStorage:** armazena uma informação no browser de forma permanente. Ao contrário da *sessionStorage*, a *localStorage* mantém o dado salvo mesmo se a aba ou a janela do navegador for fechada. A forma de proteção da informação é feita por domínio, ou seja, o domínio do site em que o script criou a informação no localStorage é quem terá acesso a informação, isto é, *um script de outro domínio não pode acessar a informação de localStorage de outro*.

Os métodos abaixo estão disponíveis em ambas as APIs. Veja:

- **setItem(chave, valor):** salva uma informação no navegador através de uma chave (identificador).
  Ex:
  localStorage.**setItem(“cor”, “branco”)**
  sessionStorage.**setItem(“cor”, “branco”)**

- **getItem(chave):** busca uma informação salva no navegador através de uma chave.
  Ex:
  **let** corSalva = localStorage**.getItem(“cor”)**
  **let** outraCorSalva = sessionStorage**.getItem(“cor”)**

- **removeItem(“chave”):** remove uma informação salva no navegador através de uma chave.
  Ex:
  localStorage**.removeItem(“cor”)**
  sessionStorage**.removeItem(“cor”)**

- **clear():** limpa todas as informações salvas no navegador (pela aplicação atual e não de todas as aplicações) do usuário.
  Ex:

  localStorage**.clear()**
  sessionStorage**.clear()**

## Spread Operator e Parâmetros REST

O operador de propagação (spread operator) permite que um objeto iterável (Arrays, Strings, arguments, etc.) tenham seus valores atribuídos de uma só vez em parâmetros de funções ou em outros elementos (como Arrays) por exemplo.

Os REST parameters permitem que possamos definir em funções um número indefinido de parâmetros, isto é, a possibilidade de termos uma função com um número indeterminado de valores que possam ser passados via argumento.

Tanto o Operador Spread quando o REST parameters são representados pela sintaxe de 3 pontos (...)

```
const numeros = [ 3, 4, 5]
const numerosCompletos = [1, 2, ...numeros, 6, 7, 8]
console.log(numerosCompletos) // [1, 2, 3, 4, 5, 6, 7, 8]
```

No exemplo anterior, pegamos os valores do array “numeros” e os atribuímos no meio do array “numerosCompletos” usando o **Spread Operator**.

Também podemos usar o Spread Operator para passar diversos valores de uma só vez para funções que aceitam um ou mais parâmetros como a console.log(), veja:

```
console.log(1, 2, 3, 4) // Exibe 1 2 3 4

console.log(numerosCompletos) // Exibe como array [1, 2, 3, 4, 5, 6, 7, 8]

// Mostra valores separados (como no primeiro exemplo)
// Exibe: 1 2 3 4 5 6 7 8
console.log(...numerosCompletos)
```

Vejamos agora os rest parameters.
Como vimos no exemplo anterior, há algumas funções no JavaScript que aceitam um número indeterminado de parâmetros. A **console.log()** é um exemplo: você pode passar um ou vários valores para serem exibidos no console de uma só vez:

```
console.log(1, 2, 3, 4) // Exibe 1 2 3 4
```

Isso quer dizer que a função **console.log()** provavelmente implementa os rest parameters (a possibilidade de criar uma função que tenha um número indefinido de parâmetros).

Quando utilizamos *rest parameters*, recebemos esses valores sempre como um array dentro de nossa função. Vejamos um exemplo:

```
function mostrarPalavras(...palavras) {
    // todo rest parameter (neste caso "...palavras") é um array de valores
    // logo podemos percorrê-lo usando forEach ou o for...of (para iteráveis)
    for (let palavra of palavras) {
        document.write(palavra + "<br>")
    }
}

mostrarPalavras('gato', 'cachorro', 'pássaro')
mostrarPalavras("Fabi", "Josy", "Marta", "Marcia", "Ágatha")
```

Ao criarmos uma função com *rest parameters* podemos, ao chamá-la, passar uma quantidade indefinida de argumentos (1 ou mais).

Vale lembrar também que caso você precise ter uma função com parâmetros fixos obrigatórios, mas ainda sim precise utilizar *rest parameters* para os demais valores, a declaração do *rest parameter* deve ser SEMPRE a última da função.

Ex:

```
function mostrarRegiao(regiao, ...estados) {
    document.write(regiao + "<br><br>")
    estados.forEach((estado) => document.write(estado + "<br>"))
}

mostrarRegiao("Região Sul", "RS", "SC, "PR)
mostrarRegiao("Região Sudeste", "SP", "RJ", "MG", "ES")
```