

# ***FUNÇÕES (PARTE 3)***

***BIND, CALL E APPLY  
ARGUMENTS  
REST PARAMETERS  
(AULA 11)***

***CURSO BÁSICO DE PROGRAMAÇÃO COM JAVASCRIPT***

***MAYARA MARQUES***

***mmrosatab@gmail.com***

# SUMÁRIO

- Recordando...
- This
- This em funções regulares
- This em arrow functions
- Bind, call e apply
- Arguments
- Rest parameters
- Quando usar funções regulares
- Quando usar arrow functions
- Mão na massa

# RECORDANDO...

- Uma **função regular** no JavaScript é criada através da palavra reservada **function**.

```
function hello () {  
    console.log('Hello')  
}
```

# RECORDANDO...

- Uma **função de seta** no JavaScript é criada através da sintaxe de seta ( => )

```
const hello = () => {  
  console.log('Hello')  
}
```

# THIS

- **this**

- Em JavaScript, **this** é uma palavra-chave que se refere ao objeto associado ao contexto em que um trecho de código está sendo executado.
- Esse contexto depende de como e onde o código está sendo executado, podendo representar diferentes objetos em diferentes situações, como o objeto **global** ou um objeto **específico**.

# THIS EM FUNÇÕES REGULARES

- Se uma **função regular** é chamada no contexto global, o `this` se refere ao objeto global (no navegador, isso é geralmente o `window`).

```
console.log(this) // Window {window: Window, self: Window, document: document, name: '', location: Location, ...}

function whoIsTheThis() {
  console.log(this) // Window {window: Window, self: Window, document: document, name: '', location: Location, ...}
}

whoIsTheThis()
```

# THIS EM FUNÇÕES REGULARES

- Se uma **função regular** é chamada por meio de um objeto, o **this** se refere a esse objeto.

```
console.log(this) // Window {window: Window, self: Window, document: document, name: '', location: Location, ...}  
  
function whoIsTheThis() {  
  console.log(this) // {myFunction: f}  
}  
  
const anyObject = { myFunction: whoIsTheThis }  
  
anyObject.myFunction()
```

# ***THIS EM FUNÇÕES REGULARES***

- Quando utilizamos funções regulares uma ***vinculação dinâmica*** de contexto é feita.
- A vinculação dinâmica, em JavaScript e em muitas outras linguagens de programação, refere-se ao **processo pelo qual o vínculo de uma função a um objeto específico é determinado em tempo de execução**, ou seja, durante a execução do programa.



# THIS EM ARROW FUNCTIONS

- Arrow functions **NÃO** possuem seu próprio this. O valor do this **independe** do objeto que chamou a função.
- O valor do this em arrow functions é **herdado do contexto em que a arrow function foi criada**.



# THIS EM ARROW FUNCTIONS

- O valor do this em arrow functions é **herdado do contexto em que a arrow function foi criada.**

```
console.log(this) // Window {window: Window, self: Window, document: document, name: '', location: Location, ...}

const whoIsTheThis = () => {
  console.log(this) // Window {window: Window, self: Window, document: document, name: '', location: Location, ...}
}

whoIsTheThis()
```

# THIS EM ARROW FUNCTIONS

- O valor do this em arrow functions é **herdado do contexto em que a arrow function foi criada**.

```
console.log(this) // Window {window: Window, self: Window, document: document, name: '', location: Location, ...}

const whoIsTheThis = () => {
  console.log(this)
}

const anyObject = { myFunction: whoIsTheThis }

anyObject.myFunction() // Window {window: Window, self: Window, document: document, name: '', location: Location, ...}
```

# THIS EM ARROW FUNCTIONS

- As arrows functions, ou funções de seta, **NÃO** possuem seu próprio this e sofrem processo de **vinculação estática**.
- Na vinculação estática, o vínculo entre uma função e valor do this é determinado **na definição da função**, ou seja, antes da execução do código. Para saber quem é o this dentro da arrow function, é necessário **analisar o local que arrow function foi criada**.



```
const show = () => {  
  console.log(this) // Window {0: Window, window: Window, self: Window, document: document, name: '', location: Location, ...}  
}  
  
show()
```

```
const show = () => {  
  console.log(this)  
}  
  
const obj = { show: show }  
  
obj.show()
```

Qual será o valor  
do this no código  
apresentado?



```
const show = () => {  
  console.log(this) // Window {0: Window, window: Window, self: Window, document: document, name: '', location: Location, ...}  
}  
  
const obj = { show: show }  
  
obj.show()
```

# COLINHA

1. Se é uma função regular (declarada com **'function'**), devo analisar a forma como estou **chamando a função**.
  - a. Se chamo a função por meio de um objeto com o ponto (.), o this é o próprio objeto.
  - b. Se chamo a função **"solta"** (não sendo valor de uma propriedade de um objeto), o this é o objeto global (window no navegador (frontend), global no NodeJS (backend)).
2. Se é uma função de seta (arrow function), o this é o contexto de onde a função foi **definida (criada)**, não onde foi chamada. Arrow function **NÃO** tem o seu próprio this. O this é **HERDADO**.





# EXEMPLOS

```
const obj = {  
  name: 'Ana',  
  sayHi: function () {  
    console.log(this.name)  
  }  
}  
  
obj.sayHi()
```

# EXEMPLOS

```
const obj = {  
  name: 'Ana',  
  sayHi: function () {  
    console.log(this.name) // `this` foi vinculado dinamicamente - aponta para `obj`  
  }  
}  
  
obj.sayHi() // Ana
```

# EXEMPLOS

```
const obj = {  
  name: 'Ana',  
  sayHi: () => {  
    console.log(this.name)  
  }  
}  
  
obj.sayHi()
```

# EXEMPLOS

```
const obj = {  
  name: 'Ana',  
  sayHi: () => {  
    console.log(this.name) // `this` foi vinculado estaticamente - não aponta para `obj`  
  }  
}  
  
obj.sayHi() // undefined
```

# EXEMPLOS

```
const obj = {  
  name: 'Ana',  
  sayHi: () => {  
    console.log(this.name) // `this` foi vinculado estaticamente - não aponta para `obj`  
  }  
}  
  
obj.sayHi() // undefined
```

# EXEMPLOS

```
const obj = {  
  value: 42,  
  getValue: function () {  
    function show () {  
      console.log(this)  
      console.log(this.value)  
    }  
    show()  
  }  
}
```

```
obj.getValue()
```

# EXEMPLOS

```
const obj = {  
  value: 42,  
  getValue: function () {  
    function show () {  
      console.log(this)  
      console.log(this.value)  
    }  
    show()  
  }  
}
```

```
obj.getValue() //Window {0: Window, window: Window, self: Window,  
document: document, name: '', location: Location, ...}  
// undefined
```

# EXEMPLOS

```
const obj = {  
  value: 42,  
  getValue: function () {  
    const show = () => {  
      console.log(this)  
      console.log(this.value)  
    }  
    show()  
  }  
}
```

```
obj.getValue()
```



# EXEMPLOS

```
const obj = {  
  value: 42,  
  getValue: function () {  
    const show = () => {  
      console.log(this)  
      console.log(this.value)  
    }  
    show()  
  }  
}
```

```
obj.getValue() // {value: 42, getValue: f}  
              // 42
```

# EXEMPLOS

```
const obj = {  
  value: 42,  
  getValue: function () {  
    const show = () => {  
      console.log(this)  
      console.log(this.value)  
    }  
    show()  
  }  
}
```

Arrow function **NÃO** tem seu próprio this. Ela herda o this do contexto em que ela foi criada.

```
obj.getValue() // {value: 42, getValue: f}  
              // 42
```

Agora que já sabemos o conceito de `this` e sua diferença em funções de seta e funções regulares. Vejamos algumas formas de **alterar o contexto de execução utilizando funções regulares**.

# ***BIND, CALL E APPLY***

- Os métodos bind, call e apply são usados para **definir o valor “*this*” dentro de uma função.**

# ***BIND***

- Do inglês **bind** significa vincular. O javascript fornece a função **bind** para que possamos fazer o vínculo do `this(contexto)` para alguma função. Ou seja, o **bind** permite especificar um contexto para a execução de uma função.

# BIND

```
const myObject = {  
  name: "John",  
  hello: function hello() {  
    console.log("My name is: ", this.name)  
  }  
}
```

`myObject.hello()` // Output: My name is: John

# BIND

```
const myObject = {  
  name: "John",  
  hello: function hello() {  
    console.log("My name is: ", this.name)  
  }  
}
```

```
const otherObject = {  
  name: "Paul"  
}
```

```
const bindedFunction = myObject.hello.bind(otherObject)  
bindedFunction() // Output: My name is: Paul
```

# CALL

- O método **call** é usado para invocar uma função imediatamente e permite especificar o contexto (this) no qual a função será executada.
- Ele aceita argumentos separados por vírgula após o contexto (this).
- É útil quando você deseja chamar uma função com um contexto específico apenas uma vez.



# CALL

```
const person1 = {
  firstName: 'John',
  lastName: 'Doe',
}

const person2 = {
  firstName: 'Jane',
  lastName: 'Doe',
}

function greet(greeting) {
  console.log(greeting + ' ' + this.firstName + ' ' + this.lastName)
  //return greeting + ' ' + this.firstName + ' ' + this.lastName
}

console.log(greet.call(person1, 'Hello')) // undefined
console.log(greet.call(person2, 'Hi')) // Output: Hi Jane Doe
```

# APPLY

- Assim como call, apply permite especificar o contexto (this) no qual a função será executada.
- A principal diferença é que o apply **aceita argumentos como um array**.
- É útil quando você tem os argumentos da função em um array ou quando deseja passar uma lista dinâmica de argumentos.

# APPLY

```
function greet(greeting) {  
  return greeting + ' ' + this.firstName + ' ' + this.lastName  
}
```

```
const person1 = {  
  firstName: 'John',  
  lastName: 'Doe',  
  greet: greet  
}
```

```
const person2 = {  
  firstName: 'Jane',  
  lastName: 'Doe',  
  greet: greet  
}
```

```
console.log(person2.greet("Hello")) // Output: Hello Jane Doe  
console.log(person2.greet.apply(person1, ["Hi"])) // Output: Hi John Doe  
console.log(person2.greet.call(person1, "Hi")) // Output: Hi John Doe
```

# APPLY

```
function greet(greeting, despedida) {  
  return greeting + ' ' + despedida + ' ' + this.firstName + ' ' + this.lastName  
}
```

```
const person1 = {  
  firstName: 'John',  
  lastName: 'Doe',  
  greet: greet  
}
```

```
const person2 = {  
  firstName: 'Jane',  
  lastName: 'Doe',  
  greet: greet  
}
```

```
console.log(person2.greet("Hello")) // Output: Hello Jane Doe  
console.log(person2.greet.apply(person1, ["Hi", "Tchau"])) // Output: Hi John Doe  
console.log(person2.greet.call(person1, "Hi")) // Output: Hi John Doe
```

# APPLY

```
function greet(greeting1, greeting2) {  
  return greeting1 + ' ' + greeting2 + ' ' + this.firstName + ' ' + this.lastName  
}  
  
const person1 = {  
  firstName: 'John',  
  lastName: 'Doe',  
  greet: greet  
}  
  
const person2 = {  
  firstName: 'Jane',  
  lastName: 'Doe',  
  greet: greet  
}  
  
console.log(person2.greet("Hello", "Good Morning")) // Output: Hello Good Morning Jane Doe  
console.log(person2.greet.apply(person1, ["Hi", "Good Morning"])) // Output: Hi Good Morning John Doe  
console.log(person2.greet.call(person1, "Hi", "Good Morning")) // Output: Hi Good Morning John Doe
```

# ***BIND, CALL E APPLY***

Em resumo:

- Os métodos bind, call e apply são usados para definir o valor “this” dentro de uma função.
- O método bind é usado para **criar** uma nova função com um valor “this” diferente.
- Os métodos call e apply são usados para chamar uma função, especificar o valor “this” e os argumentos a serem passados para a função. O apply recebe os argumentos da função passado via **arguments** e o call recebe os parâmetros da função normalmente.

# ARGUMENTS

- O ***arguments*** é recurso que permite a passagem de vários parâmetros para o interior de função.

# ARGUMENTS

```
function sum(n1, n2, n3){  
    return n1 + n2 + n3  
}
```

```
console.log(sum(10, 5, 2)) // Output: 17
```



# ARGUMENTS

```
function sum() {  
    return Array.from(arguments).reduce((accumulator, current) => accumulator + current)  
}  
  
console.log(sum(10, 5, 2)) // Output: 17
```


# ARGUMENTS

```
function sum() {  
  return Array.from(arguments).reduce((accumulator, current) => accumulator + current)  
}  
  
console.log(sum(10, 5, 2)) // Output: 17
```

A grande vantagem de usar *arguments* é que se pode passar uma quantidade variável de valores como parâmetro.

# ARGUMENTS

```
function sum() {  
    return Array.from(arguments).reduce((accumulator, current) => accumulator + current)  
}  
  
console.log(sum(10, 10, 10, 10, 10)) // Output: 50
```



A grande vantagem de usar ***arguments*** é que se pode passar uma quantidade variável de valores como parâmetro.

# ARGUMENTS

A função **typeof** permite verificar o tipo dados de uma variável ou valor passado.

```
function sum(){  
  console.log(typeof arguments) // Output: object  
  return Array.from(arguments).reduce((accumulator, current) => accumulator + current) // Output: 50  
}  
  
console.log(sum(10, 10, 10, 10, 10))
```

O **arguments** é do tipo **object**, então muitas vezes seu tipo é convertido para array para que se possa utilizar outras operações.



# ARGUMENTS

```
function sum(){  
  console.log(arguments) // Output: Arguments(5) [10, 10, 10, 10, 10, callee: f, Symbol(Symbol.iterator):  
f]  
  return Array.from(arguments).reduce((accumulator, current) => accumulator + current) // Output: 50  
}  
  
console.log(sum(10, 10, 10, 10, 10))
```

O *arguments* é do tipo **object**, então muitas vezes seu tipo é convertido para array para que se possa utilizar outras operações.



# ARGUMENTS

- Acessando valores
  - Para acessar os valores passados para uma função usando o objeto arguments em JavaScript, você pode acessá-los diretamente através do índice dentro do objeto arguments.

# ARGUMENTS

```
function exampleFunction() {  
    // Acessando o primeiro argumento  
    console.log(arguments[0])  
  
    // Acessando o segundo argumento  
    console.log(arguments[1])  
  
    // Acessando todos os argumentos usando um loop  
    for (let i = 0; i < arguments.length; i++) {  
        console.log(arguments[i])  
    }  
}  
  
exampleFunction('hello', 123, true)  
  
// hello  
// 123  
// true  
// hello  
// 123  
// true
```

# ARGUMENTS

- Arrow functions **NÃO** possuem seu próprio arguments.
- Arrow functions **herdam o objeto arguments do contexto** em que estão sendo executadas. Isso pode levar a comportamentos inesperados em alguns casos.



# ARGUMENTS

- Aqui está um exemplo para ilustrar isso:

```
function regularFunction() {  
  console.log(arguments[0]) // Output: 1  
  const arrowFunc = () => {  
    console.log(arguments[0]) // Output: 1  
  }  
  arrowFunc(2)  
}  
  
regularFunction(1)
```

Vale a pena conhecer `arguments`, especialmente para entender código legado. No entanto, para código novo, o ideal é usar **rest parameters** (`...args`).

# REST PARAMETERS

- Uma alternativa ao objeto arguments
  - Desde o ECMAScript 6, é comum usar rest parameters (**...args**) como **uma alternativa mais flexível e expressiva ao objeto arguments**. Os rest parameters permitem capturar todos os argumentos restantes em uma array, tornando o código mais legível e fácil de entender.

# REST PARAMETERS

```
function sum() {  
  return Array.from(arguments).reduce((accumulator, current) => accumulator + current) }  
  
console.log(sum(10, 10, 10, 10, 10))// Output: 50
```

```
const sum = (...args) => {  
  return args.reduce((accumulator, current) => accumulator + current)  
}  
  
console.log(sum(10, 10, 10, 10, 10))// Output: 50
```

# QUANDO USAR FUNÇÕES REGULARES

- Cenários mais indicados para utilizar **funções regulares**
  - Funções mais longas e complexas que precisam de seu próprio valor de "this".
  - Funções que precisam ser vinculadas a um objeto específico usando o método "bind".
  - Funções que precisam de seu próprio valor de "arguments" ou "super".
  - Funções que precisam ser reutilizadas em vários lugares do código.

# QUANDO USAR ARROW FUNCTION

- Cenários mais indicados para utilizar **arrow functions**
  - Funções curtas e simples que não precisam de seu próprio valor de "this" ou do "arguments".
  - Funções que precisam ser passadas como parâmetro em outras funções(funções de callback).
  - Encadeamento de métodos usando sintaxe de ponto.



***MÃO NA MASSA***

# MÃO NA MASSA



1. Crie um objeto *person* com propriedades ***name*** e ***age***. Em seguida, defina um método *sayHello* que imprime "***Hello, {name}!***" onde ***{name}*** é o valor da propriedade *name* do objeto.
2. Use *bind* para vincular uma função a um objeto e, em seguida, chame a função vinculada para exibir propriedades desse objeto.
3. Crie uma função ***sum*** que aceita dois parâmetros e use *call* para calcular a soma de dois números passados como argumentos separados.
4. Crie um objeto ***calculator*** com métodos ***add***, ***subtract***, ***multiply*** e ***divide***. Use ***apply*** para chamar dinamicamente esses métodos com argumentos passados como um array.

## MÃO NA MASSA



5. Escreva uma função ***average*** que calcula a média dos valores passados como argumentos usando ***arguments***
6. Crie uma função ***concatenate*** que recebe uma string e usa ***arguments*** para concatenar todas as strings passadas como argumentos.
7. Escreva uma função ***sumAll*** que aceita um número variável de argumentos e retorna a soma de todos eles. OBS: Use ***rest parameters***
8. Crie uma função ***mergeObjects*** que aceita um número variável de objetos como argumentos e retorna um ***único objeto mesclando todas as propriedades***.



# REFERÊNCIAS

- <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Operators/this>
- <https://indepth.dev/posts/1357/getting-started-with-modern-javascript-variables-and-scope>
- [https://www.w3schools.com/js/js\\_this.asp](https://www.w3schools.com/js/js_this.asp)
- [https://www.w3schools.com/js/js\\_scope.asp](https://www.w3schools.com/js/js_scope.asp)
- <https://youtu.be/dWZIPlc3szg?feature=shared>
- <https://youtu.be/fVXp7ZWjIO4?feature=shared>
- <https://youtu.be/ajTvmGxWQF8?feature=shared>