O que é?

A palavra reservada **this** é uma referência de contexto.

Por exemplo, quando você declara **this** dentro de um objeto, dentro do método de um objeto.

```
const pessoa = {
  firstName : "Leonardo",
  lastName : "Machado",
  id : 17,
  sexo : "Masculino",
  cargo : "Frontend",
  fullName : function() {
    console.log(this.firstName + " " + this.lastName);
    console.log(this.sexo + " " + this.cargo);
  },
  getId: function() {
    console.log(this.id);
  }
};

pessoa.fullName();
pessoa.getId();
```

fullName é uma função, então fullName é um método, então sempre que uma função está dentro de um objeto a gente chama ela de método. Então quando você utiliza **this** dentro de um método ele vai se referir ao objeto pai desse método, que neste caso é pessoa.

Quando eu chamo pessoa.fullName e ela retorna **this**.firstName, **this** quer dizer pessoa e ela vai retornar firstName desse objeto pessoa e lastName a mesma coisa e assim por diante, a gente pode fazer isso com todas as propriedades.

Seu valor pode mudar de acordo com o lugar no código onde foi chamado

Contexto	Referência
Em um objeto (método)	Próprio objeto dono do método
Sozinha	Objeto global (em navegadores, window)
Função	Objeto global
Evento	Elemento que recebeu o evento

Quando a palavra **this** está no contexto de um objeto, quando ela é o método de um objeto, ela vai se referir ao objeto dono daquele método.

Quando ela é declarada sozinha ela vai se referir ao objeto global, ou seja, quando você está no navegador, abre o console e só escreve **this** vai logar o objeto window, porque é o objeto global, é o objeto pai de todo aquele documento.

Dentro de uma função, **this** também vai se referir ao objeto global daquela função, uma função fora do objeto.

Dentro de um evento ela vai se referir ao elemento que recebeu aquele evento.

No evento:
```
<head>
  <title>Document</title>
</head>
<body>
  <button
 id="my-btn" onclick="console.log(this)">click me! </button>
</body>
```
No console se eu mandar o **this** ele vai logar, o **this** é o elemento que recebeu o evento, neste caso ele vai se referir ao botão e aí você pode fazer um **this**. e criar algum método dentro desse botão.

Exemplos:

Fora de função: console.log(this);

Se a gente logar apenas o **this** também a gente vai ver que window é um objeto global, é o objeto pai de tudo que vem dentro do navegador então é isso que vai logar.

Dentro de uma função: (function () { console.log(this);})();

O objeto função do JS nativo ele vai ter várias funções globais como: clearTimeout, setInterval etc. Então dentro de uma função o **this** vai ser o objeto global também, como a gente já tem uma função executada, o código não está mais vazio, então o objeto global ele vai ter aquela formatação, ele vai ser o objeto global.

Dentro de um objeto (método):
```
const pessoa = {
  firstName : "Leonardo",
  lastName : "Machado",
  id : 17,
  sexo : "Masculino",
```

```
  cargo : "Frontend",
  fullName : function() {
    console.log(this.firstName + " " + this.lastName);
    console.log(this.sexo + " " + this.cargo);
  },
  getId: function() {
    console.log(this.id);
  }
};

pessoa.fullName();
pessoa.getId();
```

Etapa 2: Manipulando seu Valor

Tendo explicado qual é o valor da palavra **this,** quando ela não sofreu nenhuma modificação dentro desses contextos, a gente vai ver como nós podemos manipular o valor de **this** pra fazer com que o código seja executado de acordo com dados que a gente quer.

1° Método: Call

```
const pessoa = {
  nome: 'Leonardo',
};

const animal = {
  nome: 'Golden',
};

function getSomething() {
  console.log(this.nome);
}
getSomething.call(pessoa);
```

A função não tem nenhuma relação com nenhum desses objetos e eu coloco o console **this**.nome, mas esse **this** dentro dessa função vai se referir a pessoa, quando eu uso o método call eu coloco o objeto do qual eu quero que **this** se refira, então neste caso **this** se refere a pessoa, então ele vai exibir o meu nome e eu posso alterar de pessoa para animal.

Da mesma forma a gente também pode usar essa chamada do call quando a função tem argumentos, por exemplo se a minha função se chama soma e ela recebe 2

argumentos a e b, e o meu objeto tem 2 argumentos também, então eu posso fazer na última linha, eu coloco o objeto que vai servir de referência, então o **this** vai valer myObj e aí separado por vírgulas eu vou mandar o restante dos argumentos.

Código:

```
const myObj = {
  num1: 2,
  num2: 4,
};

function soma(a, b) {
  console.log(this.num1 + this.num2 + a + b);
}
soma.call(myObj, 5, 10); // Última linha aqui
```

Ele soma todos os valores: 2 + 4 + 5 + 10 = 12

Então, é possível passar parâmetros para essa função separando-os por vírgulas.

2° Método: Apply

Ela tem um funcionamento, uma implementação muito parecida com ao call, ela só tem uma pequena diferença, olhe o mesmo exemplo que vimos primeiro:

```
const pessoa = {
  nome: 'Leonardo',
};

const animal = {
  nome: 'Golden',
};

function getSomething() {
  console.log(this.nome);
}
getSomething.apply(pessoa);
```

Se eu chamar o apply aqui não muda nada.

A grande diferença está no 2° exemplo:

```
const myObj = {
  num1: 2,
```

```
  num2: 4,
};

function soma(a, b) {
  console.log(this.num1 + this.num2 + a + b);
}
soma.apply(myObj, [5, 10]);
```

Na hora de você passar argumentos, os argumentos do apply vão ser passados dentro de um array [] essa é a diferença, enquanto no call a gente manda os argumentos separados apenas por vírgulas, no apply a gente vai mandar os argumentos dentro de um array [].

Então, é possível passar parâmetros para essa função dentro de um array.

3° Método: Bind
Ele funciona de um jeito diferente, ele clona a estrutura da sua função onde ele foi chamado e aplica o valor do objeto passado como parâmetro.

```
const retornaNomes = function() {
        return this.nome;
}

let leonardo = retornaNomes.bind({ nome: ‘Leonardo’ });

leonardo();
```

A gente tem a função retornaNomes e basicamente ela vai retornar **this**.nome, então eu vou declarar uma variável chamada leonardo e o retornaNomes vai ser retornaNomes.bind e isso vai fazer com que a função seja clonada. E o clone dela que a gente criou pra essa referência que é a variável leonardo, vai ser executado no objeto que tem nome: ‘Leonardo’, então Leonardo vai se tornar return **this**.nome com nome sendo Leonardo. E como let leonardo ela é um clone da função ela vai ser igual a uma função, então a gente precisa chamar ela como se fosse uma função -> leonardo() e vai logar o meu nome.

É como se a variável leonardo fosse = function essa mesma coisa só que ao invés de **this**.nome ela vai pegar o valor do objeto.

Então o Bind vai clonar a estrutura da função na qual você chamá-lo e vai utilizar o valor de **this,** vai ser o objeto que você passar dentro.