Para saber mais: this, bind(), apply() e call()

Você deve ter reparado na palavra-chave this que foi utilizada anteriormente, tanto nas funções construtoras quanto nas classes, e que significa literalmente “isso” ou “este”. Vamos ver com mais detalhes o que faz o this e mais três métodos que utilizamos para manipulá-lo: call(), apply() e bind().

Vamos criar um objeto pessoa com propriedades nome e email, além de um método para imprimir o nome da pessoa no terminal:

```
const pessoa = {
 nome: "Ana",
 email: "a@email.com",
 imprimeNome: function(){
   console.log(`${pessoa.nome}`)
 }
}

pessoa.imprimeNome() //Ana
```

Veja que o método imprimeNome() faz referência ao próprio objeto pessoa. O JavaScript tem uma forma melhor de fazer isso, que não “acopla” o método ao nome do objeto:

```
const pessoa = {
 nome: "Ana",
 email: "a@email.com",
 imprimeNome: function(){
   console.log(`${this.nome}`)
 }
}

pessoa.imprimeNome() //Ana
```

Podemos visualizar melhor a utilização do this para “desacoplar” o método do objeto com um outro exemplo:

```
function imprimeNomeEmail(){
 console.log(`nome: ${this.nome}, email ${this.email}`)
}
```

Acima temos uma função que não está ligada a nenhum objeto. Vamos ver como podemos utilizá-la para objetos diferentes de forma independente:

```
const pessoa1 = {
 nome: "Ana",
 email: "a@email.com",
```

```
 imprimeInfo: imprimeNomeEmail
}

const pessoa2 = {
 nome: "Paula",
 email: "p@email.com",
 imprimeInfo: imprimeNomeEmail
}

pessoa1.imprimeInfo()
//nome: Ana, email a@email.com
pessoa2.imprimeInfo()
//nome: Paula, email p@email.com
```

Dentro de cada objeto, criamos um método que chama a função externa imprimeNomeEmail(). Essa função é executada no **contexto** de cada um dos objetos e o this faz com que ela utilize os valores de propriedade desses objetos.

O this representa o objeto que executa a função. Podemos dizer que a instrução passada para o JavaScript é: "execute a função aqui, utilizando este contexto".
No caso de atributos das funções construtoras ou dos construtores de classe, o this tem função similar: podemos dizer que o construtor recebe os atributos **deste** objeto:

```
function Pessoa(nome, email){
 this.nome =nome
 this.email =email
 this.imprimeNomeEmail = function(){
    console.log(`nome: ${this.nome}, email: ${this.email}`)
 }
}
```

Quando criarmos objetos com base nesta função construtora, os atributos que a função receber como parâmetro (nome e email) serão definidos no contexto de cada um deles:

```
const pessoa1 = new Pessoa("Ana", "a@email.com")
const pessoa2 = new Pessoa("Paula", "p@email.com")

pessoa1.imprimeNomeEmail()
// nome: Ana, email: a@email.com
pessoa2.imprimeNomeEmail()
// nome: Paula, email: p@email.com
```

É possível manipular os valores de this e o JavaScript tem três métodos para isso:

call()

Esse método permite que uma função seja chamada com parâmetros e valor de this específicos. Vamos ver um exemplo:

```
function imprimeNomeEmail(tipoCliente){
 console.log(`${tipoCliente} - nome: ${this.nome}, email: ${this.email}`)
}

const cliente1 = {
 nome: "Carlos",
 email: "c@email.com"
}

const cliente2 = {
 nome: "Fred",
 email: "f@email.com"
}
```

Criamos uma função imprimeNomeEmail que recebe como parâmetro um dado que chamamos de tipoCliente. Esta função imprime no terminal um string com tipoCliente mais duas informações que estão associadas a algum objeto (ainda não informado) com this.

Vamos executar a função com call():

```
imprimeNomeEmail.call(cliente1, "cliente especial")
// cliente especial - nome: Carlos, email: c@email.com

imprimeNomeEmail.call(cliente2, "cliente estudante")
// cliente estudante - nome: Fred, email: f@email.com
```

Como a função está sendo chamada como objeto do método call(), podemos especificar que o contexto de this em cada chamada se refere a um objeto diferente (cliente1 e cliente2), sem a necessidade de adicionar a função em cada um dos objetos.

O primeiro parâmetro do método call() se refere ao objeto que será usado como contexto do this e, do segundo parâmetro em diante, são passados os argumentos que a função deve receber. No caso acima, há somente um parâmetro, a string tipoCliente.

apply()

O método apply() funciona de forma muito semelhante ao call(), porém recebe os argumentos em um array ao invés de separados:

```
function imprimeNomeEmail(tipoCliente, agencia){
 console.log(`
   ${tipoCliente} da agência ${agencia}:
   - nome: ${this.nome}, email: ${this.email}
   `)
}

const cliente1 = {
 nome: "Carlos",
 email: "c@email.com"
}
```

```
const cliente2 = {
 nome: "Fred",
 email: "f@email.com"
}
```

Chamando imprimeNomeEmail com o método apply() e passando um array de dados como segundo parâmetro (lembrando que o primeiro parâmetro de apply() se refere ao contexto de this, igual ao método call():

```
const clienteEspecial = ["cliente especial", "Rio de Janeiro"]
const clienteEstudante = ["cliente estudante", "Fortaleza"]

imprimeNomeEmail.apply(cliente1, clienteEspecial)
// cliente especial da agência Rio de Janeiro: - nome: Carlos, email: c@email.com

imprimeNomeEmail.apply(cliente2, clienteEstudante)
// cliente estudante da agência Fortaleza: - nome: Fred, email: f@email.com
```

Utilize o método apply() caso você tenha um array de dados e o call() para passar valores individuais como parâmetro. Lembre-se que o array deve seguir a ordem correta dos parâmetros informado na função.

bind()

O método bind() "prende" ou "liga" uma função ao contexto de um objeto. Por exemplo:

```
const personagem = {
 nome: "Princesa Leia",
 apresentar: function(){
  return `a personagem é ${this.nome}`
 }
}
```

O objeto acima contém uma propriedade nome e um método apresentar que retorna um string com nome; this.nome liga a propriedade nome ao contexto do objeto em que a função está definida, ou seja, "este objeto".

Vamos ver o que acontece se tentarmos executar essa função a partir de outro contexto:

```
const personagemGenerico = personagem.apresentar
console.log(personagemGenerico())
//a personagem é undefined
```

Quando atribuímos apresentar() à variável personagemGenerico estamos retirando a função apresentar() do contexto do objeto na qual foi criada, e por isso this não está mais acessível; a função perdeu a referência original e não consegue mais localizar onde está this.

Ressolvemos este problema com bind():

```
const personagemDefinido = personagemGenerico.bind(personagem)
console.log(personagemDefinido())
//a personagem é Princesa Leia
```

Acima, utilizamos o método bind() para "ligar" a função que atribuímos a personagemGenerico ao objeto personagem. Assim, sempre que esta função for executada a partir da variável personagemDefinido, a função original vai usar o objeto personagem como contexto de execução. Dessa forma, this sempre se refere ao objeto personagem e é capaz de acessar suas propriedades.

Estes três métodos têm uma variedade enorme de usos no dia a dia da programação com JavaScript, faça mais testes a partir dos exemplos acima para se familiarizar com os conceitos.