**¿QUIÉN ES THIS?**

**This** hace referencia a un un objeto y su valor depende de donde lo usemos. Por lo tanto, this será variable y dependerá siempre de donde lo utilicemos. En resumidas cuentas, this en Javascript representa el contexto.

Lo que hay que tener claro siempre es que this siempre intenta devolver un objeto e irá tirando hacia arriba hasta encontrar un objeto que devolver.

Por ejemplo, si escribimos this dentro de una función, nos devolverá el objeto global (window), porqué una función no es un objeto.

Ahora bien. si utilizamos this dentro de un método de un objeto, this apuntará al objeto en el que se encuentra el método.

**let perro = {**

**nombre: “Max”,**

**saludar: function() { return “Hola, me llamo ${this.nombre} Guau Guau”}**

**};**

**perro.saludar();** // This apunta a perro y por lo tanto imprime “Max”

Si escribimos this dentro de un constructor, ese this apuntará al objeto instanciado, sea como una función prototipo o una clase (ECMAScript 6). Pero ojo, que si escribes this en un callback de una función de una clase o una función prototipo, this te devolverá window.

**class Persona{**

**constructor(nombre, amigos = []){**

**this.nombre = nombre;**

**this.amigos = amigos;**

**}**

**listarAmigos(){**

**this.amigos.forEach(function (amigo){**

**console.log(`Hola, mi nombre es: ${this.nombre} y soy amigo de ${amigo}`);**

**});** // El problema es que dentro de este callback, this está referenciando a cada uno de los amigos, en vez de referenciar al objeto instanciado y, por lo tanto, no podrá leer la propiedad nombre porqué no existe dentro de amigos sino dentro del objeto instanciado.

**}**

**}**

Hay varias formas para solucionar ese problema:

1. **\_this** -> La primera solución es creando una referencia de this en la función (no en el callback) para que a la hora de intentar acceder a this realmente tengamos la referencia correcta de this. En vez de que apunte a nuestra variable, que apunte al scope del objeto instanciado. **const \_this = this** .

**listarAmigos(){**

**const \_this = this;** // Aquí alojamos la referencia de this para que dentro del callback pueda acceder al scope fuera del contexto del método. Podemos llamar a la variable como queramos pero suele ser o \_this o self.

**this.amigos.forEach(function (amigo){**

**console.log(`Hola, mi nombre es: ${\_this.nombre} y soy amigo de ${amigo}`);**

// En el console.log vemos que escribimos \_this en vez de this para utilizar esa referencia de this en el contexto del scope del objeto que queremos utilizar.

**});**

**}**

2) **método .bind()** -> El método bind() crea una nueva función donde **trae al scope local de una función, el contexto externo de this** y, nos permite usar ese contexto dentro de la función.

**listarAmigos(){**

**this.amigos.forEach(function (amigo){**

**console.log(`Hola, mi nombre es: ${this.nombre} y soy amigo de ${amigo}`);**

**}.bind(this));** // Con el bind(this) como método a la función callback traemos la referencia de this en el scope externo para que dentro de esta función sepa que significa this.

**}**

3) **arrow functions** -> Ya sabemos que en Javascript siempre quieren ir facilitandonos todos los quebraderos de cabeza que nos generan este lenguaje y con esto no iba a ser menos. En ECMAScript 2015 nos introdujeron las arrow functions que además de todo lo que hemos visto que pueden hacer, tienen otra función interesante. Con las arrow functions no nos tenemos que preocupar por si en un callback this no va a coger el contexto externo de la función porqué ya de por si la arrow function hace el bind a ese scope externo.

**listarAmigos(){**

**this.amigos.forEach((amigo) => {**

**console.log(`Hola, mi nombre es: ${this.nombre} y soy amigo de ${amigo}`);**

**}** // La arrow function ya nos hace el bind y nos trae el contexto externo dentro del contexto local de la función. Podemos usar this sin que genere ningún error/conflicto.

**}**

En resumidas cuentas, hay que usar más las arrow functions xD.

Dos artículos muy recomendados para entender bien qué es this y cómo funciona:

* <https://platzi.com/fundamentos-javascript/tutoriales/this-en-diferentes-situaciones-y-su-comportamiento/>
* <https://medium.com/entendiendo-javascript/entendiendo-this-javascript-cba60c8cec8c>

**Método .bind()**

**bind()** es un método que genera otra función que trae al scope local, el contexto (scope) externo que nosotros le indiquemos. Podemos indicar this para referenciar o al scope global o al scope de un objeto instanciado, o por lo contrario le podemos pasar como contexto un objeto cualquiera que queramos usar con las propiedades que se utilizan dentro de esa función.

Antes hemos visto esto haciendo uso de .bind() para dar como referencia de this a un callback donde cogía como contexto otro this que no queríamos utilizar:

**class Persona{**

**constructor(nombre, amigos = []){**

**this.nombre = nombre;**

**this.amigos = amigos;**

**}**

**listarAmigos(){**

**this.amigos.forEach(function (amigo){**

**console.log(`Hola, mi nombre es: ${this.nombre} y soy amigo de ${amigo}`);**

**}.bind(this));**

**}**

**}**

Bien, pues con bind podemos hacer algo más además de pasar el contexto externo a un contexto local que no nos deja acceder al contexto externo. También podemos pasarle como contexto otro objeto:

**let objetoPersona = {**

**nombre: “Persona random”,**

**}**

**class Persona{**

**constructor(nombre, amigos = []){**

**this.nombre = nombre;**

**this.amigos = amigos;**

**}**

**listarAmigos(){**

**this.amigos.forEach(function (amigo){**

**console.log(`Hola, mi nombre es: ${this.nombre} y soy amigo de ${amigo}`);**

**}.bind(objetoPersona));** // En este caso le pasamos como contexto el objeto objetoPersona y nos imprimirá this.nombre como el nombre que tenemos en el objetoPersona. Representación de this.nombre con bind en objetoPersona: objetoPersona.nombre.

**}**

**}**

Ahora bien, no solo podemos cambiar el contexto de this o pasar un objeto cualquiera como this, sino que además .bind() tiene otra opción disponible. Teniendo en cuenta que bind() nos retorna una nueva función, podemos darle el mismo uso que una clousure:

**function saludar(nombre, apellido) {**

**console.log(`Hola ${nombre} ${apellido}`);**

**}**

**const saludarDanis = saludar.bind(null, “Dani”);** // Le pasamos como primer parámetro null para indicar que no tenemos necesidad de pasar ningún contexto (this);

**saludarDanis(“Apellido”);**

**Métodos .call() y .apply()**

Tenemos otros métodos que nos permiten cambiar el contexto de nuestras funciones a parte de .bind(), estas son call() y apply().

La diferencia principal entre estas dos y bind(), es básicamente que .bind() nos retorna una función que generalmente la asignamos para luego invocarla y .call() y .apply() se utilizan para invocar directamente una función con el contexto cambiado (por lo tanto no devuelven otra función sino que ejecutan la función pero con el contexto/atributos cambiados).

**.call()**

Con call vamos a poder ejecutar una función con el contexto cambiado. Para ello tendremos disponible como primer parámetro de call el contexto que queremos a la hora de ejecutar la función y, luego podemos añadir todos los parámetros de la función original. Ej:

**const alex = {**

**nombre: “Alex”,**

**apellido: “Soto”**

**}** // Este es el objeto que vamos a pasar como contexto.

**function saludar(veces, isThisChanged = false) {**

**for (let i = 0; i < veces; i++){**

**console.log(`Hola ${this.nombre} ${this.apellido}`);**

**}**

**if (isThisChanged) {**

**console.log(`¡Ojo! El contexto está cambiado. Ahora this es: ${this}}`);**

**return this;** // Devolvemos el this cambiado (básicamente devolvemos el objeto alex)

**}**

**}**

**saludar.call(alex, 4, true);** // Además del contexto podemos especificar todos los parámetros disponibles de la función saludar().

**.apply()**

apply() es exactamente lo mismo que .call() solo que los parámetros de la función a la que ejecutamos .apply() los tenemos que pasar como un array. Básicamente tenemos dos parámetros disponibles en el método: el contexto, y el array con todos los parámetros de la función que llamamos.

**saludar.apply(alex, [4, true]);** // Además del contexto podemos especificar todos los parámetros disponibles de la función saludar() pero dentro de un array a diferencia de .call().