Things to know about 'this' in JavaScript

Are you having trouble understanding the keyword 'this' in JavaScript ? Then please read what you need to know about 'this' in JavaScript below.

'this' in global scope

In this scope, **this** will return the **window** object as long as it is outside a function. Global context means you don't put it inside a function.

```
if(true) { console.log(this) // returns window object } let i = 2 while(i  10) {
```

If you run the above code, you will have a window object.

'this' inside function

When used in a function, **this** refers to the object to which the function is bound. The exception here is when you use this in a standalone function, which returns a window. Let's look at some examples.

In the following example, the **sayName** function is inside the me object. In this case, **this** refers to the object containing this function.

```
function sayName() { return `My name is ${this.name}` } const me = { name: "King
```

This is a me object , so saying **this.name** inside the **sayName** method is exactly the same as me.name.

Another way to define it is that anything on the left side of the function when called would be **this** . This means you can reuse the **sayName** function in different objects, and this will refer to a completely different context each time.

Now as we said earlier, **this** returns the **window** object when used in a standalone function. This is because a default standalone function is bound to the **window** object .

```
function talk() { return this } talk() // returns the window object
```

Calling **talk()** is the same as calling **window.talk()** . Anything to the left of the function automatically becomes this.

Also, the **this** keyword in this function behaves differently in JavaScript strict mode. This should also be noted when you are using a UI library that uses strict mode.

Use 'this' with Function.bind()

There are cases where you cannot add a function to an object as a method.

Perhaps the object is not yours and you will get it from a library. This object is mutable, so you cannot change it. In such cases, you can still run individual functions from this object using the **Function.bind()** method .

In the following example, the **sayName** function is not a method on the me object, but you still bind it with the **bind()** function :

```
function sayName() { return `My name is ${this.name}` } const me = { name: "Kings
```

Whatever object you pass into **bind()** will be used as the value of this in that function call.

In short, you can use **bind()** on any function and pass in a new context (an object). That object will override the meaning of **this** inside that function.

Use 'this' with function.call()

What if you don't want to return the whole new function, but only call it after binding it to the context? The solution here is to use **the call()** method :

```
function sayName() { return `My name is ${this.name}` } const me = { name: "Kings
```

The call() method immediately runs this function instead of returning another function.

If this function needs a parameter, you can pass it through **the call()** method . In the following example, you are passing the language to the **sayName()** function , so you can use it to return different messages according to the condition:

```
function sayName(lang) { if (lang === "en") { return `My name is ${this.name}` }
```

As you can see, you can pass any desired parameter to this function as the second argument to the call() method. You can also pass as many parameters as you want.

The apply() method is similar to **call()** and **bind()** . The only difference here is that you pass multiple arguments by separating them with commas via **call()** , while passing multiple arguments in an array with **apply()** .

In a nutshell, bind(), call(), and apply() all allow you to call a function with a completely different object without any relationship between those two objects.

'this' in constructor function

If you call a function with the key keyword, it creates a this object and returns it:

```
function person(name){ this.name = name } const me = new person("Kingsley") const
```

In the above code, you have created 3 different objects from the same ha. **The new** keyword automatically creates a link between the object being created and the **this** keyword inside this function.

'this' in the callback function

The callback function is different from other normal functions. They are functions that you pass to another function as an argument, so they can be executed immediately after the main function has finished executing.

The this keyword refers to a completely different context when used in a callback function:

```
function person(name){ this.name = name setTimeout(function() { console.log(this
```

After a second of calling the person constructor and creating a new me object, it will record the window object as the value of this. Therefore, when used in a callback function, **this** refers to the window object and not the 'constructed' object.

You have two ways to fix this problem. The first method is to use **bind()** to bind **the person** function to the newly constructed object:

```
function person(name){ this.name = name setTimeout(function() { console.log(this
```

With the above edit, **the this** in **the callback** will point to **this** just like the constructor.

The second way to deal with the this problem in the callback function is to use arrow functions.

'this' inside arrow functions

Arrow functions are different from regular functions. You can turn the callback function into an arrow function. With this function you no longer need **bind()** as it automatically binds to the newly constructed object:

```
function person(name){ this.name = name setTimeout(() => { console.log(this) }, 
```

Here's what you need to know about the 'this' keyword and what it means in all different contexts in JavaScript. If you are new to JavaScript programming, the above knowledge is very useful to you.

You finished reading the article "**Things to know about 'this' in JavaScript**" edited by the **TipsMake** team. We hope this article has provided you with many useful tech tips and tricks. You can search for similar articles on tips and guides. Thank you for reading and for following us regularly.