Javascript: Uncovering mysteries of ‘this’ keyword

Feb 13, 2018 #Javascript

'This' is a very straightforward concept in other languages. Not so in javascript. It can point to pretty much anything depending on the context.

What the hell is *this*?

If you are coming from another language, which is Object Oriented, such as Java, you are no doubt familiar with the concept of *this*. *This* keyword is used inside of a class and refers to the current instance. No more, no less.

In Javascript, *this* is yet another concept, which behaves in an unexpected way. Like other concepts such as new operator or classes, it tricks you into making some false assumptions based on concepts which you know from other languages. Because the syntax and naming are the same or very similar, yet the concept is different. What's more, its behavior is not the same depending on whether you are in the strict mode or not.

Every time a function is invoked, *this* is assigned a reference to an object based on how the function was called. Is it a regular function? A method called on an object? Fat arrow function? Method of an ES6 class? That's the key to understanding the topic. What's confusing about this is that the very same function can have a different value of *this* based on circumstances. Turns out that this does not depend on the function itself but rather on how the function is called. The same function will have different *this* when called as a method of an object and when provided as a callback function. You need to be careful because the functions can be passed around and in such case, *this* may be different from what you expected.

Function invocation

When calling a function *this* is referring to the global object. The global object depends on how you run your code -- if you are executing in a browser, it is the window object. In node.js it is an object called global.

```
function logThis() {
  console.log(this);
}

logThis(); // global object - e.g. window
```

Note that it applies to function calls, where the function is not called as a property of an object. eg. *object.function()*. In such cases, it is actually method invocation, which follows different rules (see below).

Alright, so far so good. *This* in functions calls points to the global object. It is not so simple though. This behavior only applies when not in strict mode. When in strict mode, *this* in functions is undefined.

```
function logThisInStrictMode() {
  "use strict";
  console.log(this);
}

logThisInStrictMode(); // undefined
```

Method invocation

A method is basically a function, which is a property of an object. In methods, the situation is simple -- *this* refers to the object the method is owned by.

```
var john = {
  name: 'John',
  greet: function() {
      // this points to the enclosing object
      console.log('Hi! My name is ' + this.name);
  }
};

john.greet(); //methods are called through their object (john here)
```

Be careful though. That does not apply to nested functions. If you declare a function inside an object's method, its this does not point to the original method's object. It is a regular function, therefore, it's this points to the global object (see above). Unless, of course, you are in strict mode. In such case it is undefined.

```
var john = {
    name: 'John',
    greet: function(people) {
        people.forEach(function (person) {
            // this points to the global object
            console.log('Hi ' + person + '. My name is ' + this.name);
        });
    }
}

john.greet(['Jane', 'James', 'Jill']); // Hi Jane. My name is undefined ...
```

Alright, not good. But what can we do to fix it? We need *this* to point to the john object! There are some ways solve this issue. One common approach is storing *this* reference in a variable (by convention usually called *that*). Since functions have access to the variables declared in their outer function, we can then easily access *that* variable from the inner function. And it will still keep the original *this* which we need.

```
var john = {
    name: 'John',
    greet: function(people) {
        var that = this;
        people.forEach(function (person) {
            // this points to the global object
            console.log('Hi ' + person + '. My name is ' + that.name);
        });
    }
};

john.greet(['Jane', 'James', 'Jill']);
```

This works, but there are more solutions, which we'll cover later. For now, keep in mind that *this* points to the owner object ONLY when you call the method through the object:

```
john.greet(); //finally works
```

If you take the very same method and call it in a different way, this will be different. Let's say we store the method to a variable and then invoke it. Suddenly, when it is not called on an object, it is just an ordinary function and, as we know, the function's this points to the global object.

```
var greetingFunction = john.greet;
greetingFunction(); // Hi! My name is undefined
```

In constructor functions

When you are using constructor functions with a new keyword (https://www.vojtechruzicka.com/javascript-constructor-functions-and-new-operator/) , *this* behaves a bit differently than usual. In short, what *new* operator does is that:

1. It creates a new blank object.
2. It makes *this* to point to this newly created object inside the constructor function
3. It sets the prototype of the newly created object to the constructor function's prototype.
4. It makes the constructor function return the newly created object IF it is not returning anything.

Long story short, when using the new operator, *this* inside the constructor function points to the newly created object. Keep in mind that if you forget to include *new* operator and just call the function directly, this will not point to the new object but rather to the global object.

```
var Person = function (firstName, lastName) {
  //If you called this using new operator, this links to the new object
  this.firstName=firstName;
  this.lastName=lastName;
};
var john = new Person('John', 'Doe');
```

In classes

When using *this* inside of methods of an ES6 class, it points to the current object. That's nice.

```
class Person {
  constructor (name) {
    this.name=name;
  }

  greet() {
    console.log('Hi! My name is' + this.name);
  }
}

var john = new Person('John Doe');
john.greet(); // Hi! My name is John Doe
```

However, when using nested functions inside such methods, they behave as regular functions. Same as nested functions in object methods. There is one caveat though. The code inside classed is automatically executed in strict mode even if it is not explicitly declared! As we already know, strict mode changes behavior inside functions so this is undefined instead of pointing to the global object.

In fat arrow functions

In short, fat arrow functions are a more concise way of writing function expressions introduced in ES6. Their biggest advantage is, however, not saving a few characters when typing. They greatly simplify usage of *this*. The thing is that unlike function expressions, where *this* points to the global object, in fat arrow functions *this* is taken from the enclosing function. Remember section about methods and how we used trick *var that = this* to preserve original *this* of the other function? Arrow functions are shorter and you don't need to worry about such tricks.

```
//The old way using function expressions
var john = {
    name: 'John',
    greet: function(people) {
        var that = this;
        people.forEach(function (person) {
            // this points to the global object
            console.log('Hi ' + person + '. My name is ' + that.name);
        });
    }
};

//The new ways using fat arrow functions
var john = {
    name: 'John',
    greet: function(people) {
        people.forEach(person => {
            // this points to this of the greet function!
            console.log('Hi ' + person + '. My name is ' + this.name);
        });
    }
};

john.greet(['Jane', 'James', 'Jill']); // Hi Jane. My name is John ...
```

Setting this directly - apply, call, bind

Sometimes it is handy to be able to directly set the value of *this* when calling a function. Fortunately, there are three methods in javascript exactly for that. All of them are available to all the functions as they are on the function's prototype. They are: *apply*, *call* and *bind*.

Apply and *call* are very similar to each other. They allow you to invoke their function with a value of this specified as a parameter and by providing arguments to be passed to the function call. The value returned is the value returned by the function called. The only way in which they differ is that one expects the function arguments as an array and the other one as direct arguments. Let's compare.

```
function myFunction(arg1, arg2, arg3) {
    console.log(this)
}

var valueOfThis = {name: 'John Doe'};

//regular function invocation, this points to the global object
myFunction('arg1', 'arg2', 'arg3');

// You can call the function and specify a value of this to be used
myFunction.apply(valueOfThis, ['arg1', 'arg2', 'arg3']); // arguments passed as
myFunction.call(valueOfThis, 'arg1', 'arg2', 'arg3'); // arguments passed separ
```

Bind is different. *Bind* does not invoke the function immediately. It returns function, which has *this* bound to the value provided. That is useful in many situations. For example, when you have a method of an object and you want to pass it somewhere else as a callback function. Without its object, it is an ordinary function and when invoked this points to the global object. You can, however, pass a version of your method where you bind your original object as this. No matter where your method ends and who will use it, you can be sure that *this* will have a proper value.

```
var john = {
    name: 'John',
    greet: function (people) {
        console.log('Hi! My name is ' + this.name);
    }
};

//When we pass a function like that, this is not preserved
var greetFunction = john.greet;
greetFunction(); // Hi! My name is undefined

//Let's bind our desired value of this using bind method
var boundGreetFunction = john.greet.bind(john);
boundGreetFunction(); // Hi! My name is John
```

Conclusion

Understanding how *this* works in Javascript under various circumstances is crucial and prevents a lot of confusion and headache. Keep in mind that this depends not on the function itself but rather on how the function is called. Since functions are first-class citizens in Javascript, they can be passed around and *this* will change accordingly. It is, therefore, useful to remember to prevent possible errors with actions such as:

- Using strict mode to prevent accidental mutation of the global object
- Binding this when passing methods around
- Using arrow functions
- Using classes instead of constructor functions to prevent accidental omission of the new operator

---

Let's connect