Demystifying the JavaScript this Keyword

Summary: in this tutorial, you will learn about the JavaScript `this` value and understand it clearly in various contexts.

If you have been working with other programming languages such as Java, C#, or PHP, you're already familiar with the `this` keyword.

In these languages, the `this` keyword represents the current instance of the class and it is only relevant within the class.

JavaScript also has `this` keyword. However, the `this` keyword in JavaScript behaves differently from other programming languages.

In JavaScript, you can use the `this` keyword in the global and function contexts. Moreover, the behavior of the `this` keyword changes between strict and non-strict modes.

What is this keyword

In general, the `this` references the object of which the function is a property. In other words, the `this` references the object that is currently calling the function.

Suppose you have an object `counter` that has a method `next()` . When you call the `next()` method, you can access the `this` object.

```
let counter = {
  count: 0,
  next: function () {
    return ++this.count;
  },
};

counter.next();
```

Inside the `next()` function, the `this` references the `counter` object. See the following method call:

```
counter.next();
```

The `next()` is a function that is the property of the `counter` object. Therefore, inside the `next()` function, the `this` references the `counter` object.

Global context

In the global context, the `this` references the global object, which is the `window` object on the web browser or `global` object on Node.js.

This behavior is consistent in both strict and non-strict modes. Here's the output on the web browser:

```
console.log(this === window); // true
```

If you assign a property to `this` object in the global context, JavaScript will add the property to the global object as shown in the following example:

```
this.color= 'Red';
console.log(window.color); // 'Red'
```

Function context

In JavaScript, you can call a function in the following ways:

- Function invocation
- Method invocation
- Constructor invocation
- Indirect invocation

Each function invocation defines its own context. Therefore, the `this` behaves differently.

1) Simple function invocation

In the non-strict mode, the `this` references the global object when the function is called as follows:

```
function show() {
   console.log(this === window); // true
}

show();
```

When you call the `show()` function, the `this` references the global object, which is the `window` on the web browser and `global` on Node.js.

Calling the `show()` function is the same as:

```
window.show();
```

In the strict mode, JavaScript sets the `this` inside a function to `undefined`. For example:

```
"use strict";

function show() {
    console.log(this === undefined);
}

show();
```

To enable the strict mode, you use the directive `"use strict"` at the beginning of the JavaScript file. If you want to apply the strict mode to a specific function only, you place it at the top of the function body.

Note that the strict mode has been available since ECMAScript 5.1. The `strict` mode applies to both function and nested functions. For example:

```
function show() {
    "use strict";
    console.log(this === undefined); // true

    function display() {
        console.log(this === undefined); // true
    }
    display();
}

show();
```

Output:

```
true
true
```

In the `display()` inner function, the `this` also set to `undefined` as shown in the console.

2) Method invocation

When you call a method of an object, JavaScript sets `this` to the object that owns the method. See the following `car` object:

```
let car = {
    brand: 'Honda',
    getBrand: function () {
        return this.brand;
    }
}
```

```
console.log(car.getBrand()); // Honda
```

In this example, the `this` object in the `getBrand()` method references the `car` object.

Since a method is a property of an object which is a value, you can store it in a variable.

```
let brand = car.getBrand;
```

And then call the method via the variable

```
console.log(brand()); // undefined
```

You get `undefined` instead of `"Honda"` because when you call a method without specifying its object, JavaScript sets `this` to the global object in non-strict mode and `undefined` in the strict mode.

To fix this issue, you use the `bind()` method of the `Function.prototype` object. The `bind()` method creates a new function whose the `this` keyword is set to a specified value.

```
let brand = car.getBrand.bind(car);
console.log(brand()); // Honda
```

In this example, when you call the `brand()` method, the `this` keyword is bound to the `car` object. For example:

```
let car = {
    brand: 'Honda',
    getBrand: function () {
        return this.brand;
    }
}

let bike = {
    brand: 'Harley Davidson'
}

let brand = car.getBrand.bind(bike);
console.log(brand());
```

Output:

```
Harley Davidson
```

In this example, the `bind()` method sets the `this` to the `bike` object, therefore, you see the value of the `brand` property of the `bike` object on the console.

3) Constructor invocation

When you use the `new` keyword to create an instance of a function object, you use the function as a constructor.

The following example declares a `Car` function, and then invokes it as a constructor:

```
function Car(brand) {
    this.brand = brand;
}

Car.prototype.getBrand = function () {
    return this.brand;
}

let car = new Car('Honda');
console.log(car.getBrand());
```

The expression `new Car('Honda')` is a constructor invocation of the `Car` function.

JavaScript creates a new object and sets `this` to the newly created object. This pattern works great with only one potential problem.

Now, you can invoke the `Car()` as a function or as a constructor. If you omit the `new` keyword as follows:

```
var bmw = Car('BMW');
console.log(bmw.brand);
// => TypeError: Cannot read property 'brand' of undefined
```

Since the `this` value in the `Car()` sets to the global object, the `bmw.brand` returns `undefined` .

To make sure that the `Car()` function is always invoked using constructor invocation, you add a check at the beginning of the `Car()` function as follows:

```
function Car(brand) {
    if (!(this instanceof Car)) {
        throw Error('Must use the new operator to call the function');
    }
    this.brand = brand;
}
```

ES6 introduced a meta-property named `new.target` that allows you to detect whether a function is invoked as a simple invocation or as a constructor.

You can modify the `Car()` function that uses the `new.target` metaproperty as follows:

```
function Car(brand) {
    if (!new.target) {
        throw Error('Must use the new operator to call the function');
    }
    this.brand = brand;
}
```

4) Indirect Invocation

In JavaScript, functions are first-class citizens. In other words, functions are objects, which are instances of the Function type.

The `Function` type has two methods: `call()` and `apply()` . These methods allow you to set the `this` value when calling a function. For example:

```
function getBrand(prefix) {
    console.log(prefix + this.brand);
}

let honda = {
    brand: 'Honda'
};
let audi = {
    brand: 'Audi'
};

getBrand.call(honda, "It's a ");
getBrand.call(audi, "It's an ");
```

Output:

```
It's a Honda
It's an Audi
```

In this example, we called the `getBrand()` function indirectly using the `call()` method of the `getBrand` function. We passed `honda` and `audi` object as the first argument of the `call()` method, therefore, we got the corresponding brand in each call.

The `apply()` method is similar to the `call()` method except that its second argument is an array of arguments.

```
getBrand.apply(honda, ["It's a "]); // "It's a Honda"
getBrand.apply(audi, ["It's an "]); // "It's a Audi"
```

Arrow functions

ES6 introduced a new concept called the arrow function. In arrow functions, JavaScript sets the `this` lexically.

It means the arrow function does not create its own execution context but inherits the `this` from the outer function where the arrow function is defined. See the following example:

```
let getThis = () => this;
console.log(getThis() === window); // true
```

In this example, the `this` value is set to the global object i.e., `window` in the web browser.

Since an arrow function does not create its own execution context, defining a method using an arrow function will cause an issue. For example:

```
function Car() {
  this.speed = 120;
}

Car.prototype.getSpeed = () => {
  return this.speed;
};

var car = new Car();
console.log(car.getSpeed()); // 👉 undefined
```

Inside the `getSpeed()` method, the `this` value reference the global object, not the `Car` object but the global object doesn't have a property called speed. Therefore, the `this.speed` in the `getSpeed()` method returns `undefined` .