This repository Search Pull requests Issues Gist

Propulsion-Academy / fswd-2017-01 Private

Watch 5 Star 0 Fork 1

Code Issues 0 Pull requests 0 Projects 0 Wiki Pulse Graphs

Branch: master fswd-2017-01 / js-fundamentals / day3 / this.md Find file Copy path

Llorenç Muntaner first week 08934d7 7 days ago

0 contributors

281 lines (197 sloc) 8.29 KB Raw Blame History

`this` and context

In Object Oriented Programming there is a way to reference the actual instance. In order to do that, we have been using the keyword `this` in our cases. However, `this` is not always what we expect.

In JavaScript, `this` is the current execution context of a function.

The value of `this` is not set when we define the function, it depends on how that function is called.

There are 4 ways we can call a function.

- **Function style**: `play()`.
- **Method style**: `trivial.play()`.
- **Constructor style**: `new Trivial()`.
- **Indirect invocation**: using `.call` or `.apply`.

Each of these ways of calling a function sets `this` to a different value. This means that the same function, called in two different ways, will have a different value of `this` at call time.

Function style

What differentiate a Function style call is that there is nothing at the left side of the function. It's a simple call. The function is not attached to an object.

```
function play() {
  console.log(this);
  console.log('playing');
}

play();
```

As you can see, the call to the function is just `play()`. Nothing on the left side. Plain execution.

In this case, the value of `this` inside the function will be the main context. If you do this in the browser, it will be the `window` object. In `node` it will be the `global` object.

Method style

This is also a common way of invocation. Used mainly in Object Oriente Programming. The name of the function will be preceded by the name of the object.

```
function play() {
  console.log(this);
  console.log('playing');
}

var cat = {
  name: 'Fante',
  doSomething: play
};

play(); // function style
cat.doSomething(); // method style
```

Remember that functions are values. They can be passed around just like string, arrays, ...

Now, the function `play` is called with `cat` on the left side. Even though the function is defined the same way as before.

At the moment of the call, `play` function is bound to `cat`. Which means that the value of `this` inside the function will be the object. In this case `cat`.

It is thanks to this feature of the keyword `this` that we can create methods in classes that have access to the properties and other methods of the class.

Remember that I could also do it this other way:

```
function play() {
  console.log(this);
  console.log('playing');
}

var cat = {
  name: 'Fante'
};
cat.doSomething = play;

cat.doSomething();
```

This code is exactly the same as the previous one.

Constructor style

This is when the function is used to create an instance of a class with the keyword `new`.

```
var Writer = function(name) {
  this.name = name;
}

var fante = new Writer('Fante');
```

What happens when using the keyword `new`, is that a new object `{}` will be created and assigned to `this` so that it can be used inside the constructor.

Then that newly created object will be returned by the function. Even though it's not there, there is a `return this;` in the constructor.

There is one more thing that the keyword `new` does. It sets the prototype of the newly created object to the prototype of the constructor. We will go deeper into this in the Advanced JS week.

Indirect invocation

Last but not least, we can use methods on `Function` object to call a function.

This sounds weird, but there might be some situation where this might come handy.

There are two methods which allow us to call the function: `.apply` and `.call`.

```
function play() {
  console.log(this);
  console.log('playing');
}

var cat = {
  name: 'Fante'
};

play.call(cat); // same as play.apply(cat)
```

We are not directly calling `play()`, we are calling the method `.call` passing it a parameter. In this case the parameter passed to `call` is `cat`.

`.call` will call the function and set the value of `this` inside that function, to whatever parameter we pass to `call`. In this specific case to `cat`.

```
function play() {
  console.log(this);
  console.log('playing');
}

var cat = {
  name: 'Fante'
};

play.call(cat);

var dog = {
  name: 'Hornby'
};

play.call(dog);
```

Read about the differences between `call` and `apply` here

Common pitfalls

The keyword `this` is source of a lot of bugs and many pitfalls. Understanding how it works and how to use it will save you a lot of time debugging.

Let's start with a simple example:

```
var writer = {
  name: 'Bukowski',
  books: ['Post Office', 'Factotum', 'Women', 'Hollywood'],
  printBooks: function() {
    this.books.forEach(function(book) {
      console.log(this.name + 'has written ' + book);
    });
  }
}

writer.printBooks();
```

The previous code looks good. However this is what you get:

```
undefined has written Post Office
undefined has written Factotum
undefined has written Women
undefined has written Hollywood
```

Why is `this.name` `undefined` ? Where are we using `this.name` ?

We are using it inside `printBooks` , which is called Method Style: `writer.printBooks()` . Isn't it?

Let's be more precise. `this.name` is actually inside the function that we pass to `forEach` . That anonymous function is the callback passed to `forEach` . How is that function called? Where is it called?

The most important aspect to understand is that we are not calling the function that uses `this.name` . `forEach` is calling it.

When we pass a callback to a function, it will always be called **function style**. That means that the function we pass to `forEach` won't have the `keyword` set to the value we want. Which is the object.

Just to make it clearer, this is another way of writing the previous code.

```
var writer = {
  name: 'Bukowski',
  books: ['Post Office', 'Factotum', 'Women', 'Hollywood'],
  printBooks: function() {
    console.log(this); // in here, `this` has the expected value, since printBooks is called method style
    var printBook = function(book) {
      console.log(this.name + 'has written ' + book);
    };
    this.books.forEach(printBook);
  }
}

writer.printBooks();
```

We define `printBook` , but we never call it. We just pass it to `forEach` .

What can we do to use `this.name` inside the function that we want to pass to `forEach` ?

There are many approaches. We will explain 2 of the most common.

- Use the scope of the function:

```
var writer = {
  name: 'Bukowski',
  books: ['Post Office', 'Factotum', 'Women', 'Hollywood'],
  printBooks: function() {
    var writer = this; // we create a new variable and set the value to `this`
    var printBook = function(book) {
      console.log(writer.name + 'has written ' + book); // inside our function we use the previously created variable
    };
    this.books.forEach(printBook);
  }
}

writer.printBooks();
```

- Bind the function

```
var writer = {
  name: 'Bukowski',
  books: ['Post Office', 'Factotum', 'Women', 'Hollywood'],
  printBooks: function() {
    var writer = this; // we create a new variable and set the value to `this`
    var printBook = function(book) {
      console.log(this.name + 'has written ' + book);
    };
    this.books.forEach(printBook.bind(writer));
  }
}

writer.printBooks();
```

`.bind` is a method on `Function.prototype`. It created a new function, where the keyword `this` will have the value of the first paramenter when calling `.bind`.

If you realize it, there is a redundancy in the code. Which makes it -hopefully- slightly more friendly.

```
var writer = this; // we create a new variable and set the value to `this
// ...
this.books.forEach(printBook.bind(writer));
```

Should be just:

```
this.books.forEach(printBook.bind(this));
```

Which means we have:

```
var writer = {
  name: 'Bukowski',
  books: ['Post Office', 'Factotum', 'Women', 'Hollywood'],
  printBooks: function() {
    var printBook = function(book) {
      console.log(this.name + 'has written ' + book);
    };
    this.books.forEach(printBook.bind(this));
  }
}

writer.printBooks();
```

I know this sounds confusing. But during the following days you will fall into this pitfall a number of times. Try to keep this in your head and slowly use it to solve the bugs. With some time it will come naturally to you.

Read more

Follow up this notes with this great article