Deepak Gupta Follow
Jan 7, 2019 · 7 min read ★ · Listen

Understanding Javascript ‘this’ keyword (Context)

Photo by sydney Rae on Unsplash

What is context?

Context is always the value of the `this` keyword which is a reference to the object that “owns” the currently executing code or the function where it’s looked at.

```
> this
◂ ▶ Window {postMessage: f, blur: f, focus: f, close: f, parent: Window, …}
> this === window
◂ true
>
```

We know that `window` is a global object in the browser so if we type `this` in the console and it should return window object, which it does.

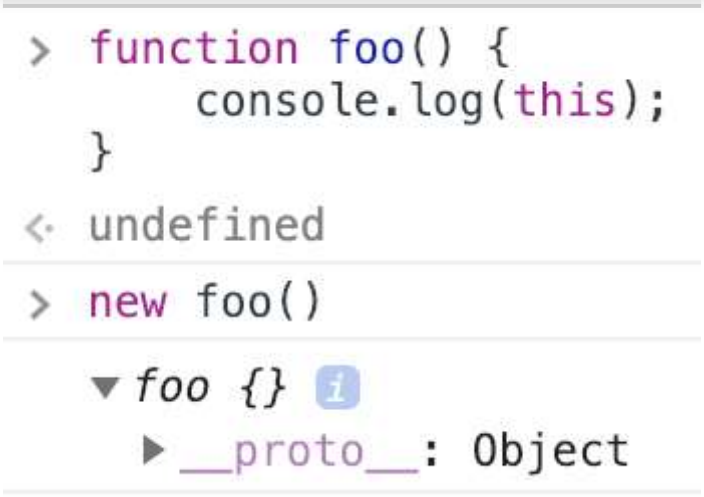
In node.js CLI if you try doing the above you will get an object that will have all globally used function like `console` , `process` etc. (try once).

> *Note: The value of `this` keyword depends on the object the function is run/called /sit on. Therefore `this` keyword has different values depending on where it is used.*
>
> *Note: From now, this and context is used interchangeably.*

Context — globally and inside a function.

```
> function foo() {
      console.log(this);
  }
◂ undefined
> foo()
  ▶ Window {postMessage: f, blur: f, focus: f, close: f, parent: Window, …}
◂ undefined
> window.foo()
  ▶ Window {postMessage: f, blur: f, focus: f, close: f, parent: Window, …}
◂ undefined
>
```

this at the global level called on the global object



this inside function called on function object at global level

Whereas if we do `new foo()` at the global level then will get `this` as `foo` object.

> *Note:* `new` *operator creates an instance of an object. Context of the function will be set to the created instance of an object.*

Context — under 2nd level function

```
> var x = {
      fn: function() {
          return this;
      },
      y : {
          fn: function() {
              return this;
          }
      }
  }
<· undefined
> x.fn() === x
<· true
> x.y.fn() === x
<· false
> x.y.fn() === x.y
<· true
> |
```

Context — when the function is defined globally and used under an object (Implicit Binding).

```
> function func () {
      return this;
  }

  var obj = {
      method: func
  };
<· undefined
> obj.method() == obj
<· true
> obj.method() == window
<· false
> |
```

> *Note: From above, we get that value of `this` keyword depends on the function is called upon not where the function is defined.*

How context behave in 'use strict`?

When using `use strict` in a function, the context i.e this keyword behaves differently. Context remains whatever it was called upon.

```
> function func () {
    'use strict';
    return this;
  }
<· undefined
> func() === window
<· false
> func() === undefined
<· true
```

> *Note: Our entire program should probably either be* `strict` *or* `non-strict`*. However, sometimes you include a third-party library that has different Strict’ness than your own code, so care must be taken over these subtle compatibility details.*

How context behave in arrow function?

Arrow functions work differently from regular functions in terms of context. `this` will always refer to the lexical scope (read here about scope), i.e `this` retains the value of the enclosing lexical context's.

```
> const a = () => {
      return this;
  }
<· undefined
> a() === window
<· true
>
```

In global code, it will be set to the global object, hence we get above true.

How does context behave on the object’s prototype chain?

Context follows the same rule, i.e. if the function is on an object’s prototype chain, `this` refers to the object the method was called on.

```
> var obj = {
      func: function() {
          return this.x;
      }
  }
< undefined
> var newObj = Object.create(obj);
< undefined
> newObj.x = 10;
< 10
> newObj.func()
< 10
> obj.func()
< undefined
```

If we call `obj.func()` will get `undefined` and if `func` is called on `newObj` created from `obj` which has `x` defined it will return the value hence 10.

How context behave in the event handlers?

The context in case event handlers refers to the **element** that received the event.

```
> $('body').on('click', function (e) {
    console.log(this)
    console.log(this === window)
    console.log(e.currentTarget === this)
  })
< ▶ jQuery.fn.init [body.document, prevObject: jQuery.fn.ini
  ▶ <body data-slug="Web/API/EventTarget/addEventListener" c
  false
  true
```

How does the context behave in an execution context?

If you don’t know what is execution context (read here). In short, execution context is the ‘environment’ or scope in which a function executes in. Every time a function is called, a new `execution context` is created. Every call to an `execution context` has 2 stages

1. Creation — when the function is called
2. Activation — when the function is executed

The value of `this` is determined at creation phase, not at the time of execution. However, `this` determination rule remains the same.

How is context is different from the scope?

Scope and context are altogether a different concept but usually used by the upcoming developer interchangeably.

The scope is the accessibility of variables, functions, or objects in some particular part of your code during runtime. Read more here about scopes.

Every function invocation has both a scope and a context associated with it.

How to explicitly change the context?

We can dynamically change the context of any method by using either `call()` , `apply()` and `bind()` method.

Call — The very first argument `call` takes in is the **context** you want to use. Afterward, you can pass in **any number of comma-separated values.**

```
foo.call(context, param1, param2, param3 );
```

Apply — This is the same as `call` but differs in the sense of no. of argument. Apply only support 2 arguments, **context and array of values**.

Bind — It returns a new function which is permanently bound to the first argument of `bind` regardless of how the function is being used. `bind` doesn’t invoke the bound function immediately, rather it returns a new function we can run later.

```
> var obj = {name: 'test'};
<· undefined
> var foo = function(a) {
      return this.name + a;
  };
<· undefined
> foo.call(obj, 'er')
<· "tester"
> foo.apply(obj, ['ing'])
<· "testing"
> var bar = foo.bind(obj)
<· undefined
> bar('er')
<· "tester"
```

Why do we need to explicitly change the context?

1. When we need to call a function defined inside an object say `x` but on other objects say `y` we can use explicit methods to do so, to **increase reusability**.

2. **Currying and partial application** is another part where explicitly change in context is used.

3. To make **utility functions** like

```
> var add = function(x, y){return x + y;}
<· undefined
> var add2 = add.bind(null, 2);

<· undefined
> add2(2)
<· 4
```

```
> var findMax = function(a) {
      return Math.max.apply(null, a);
  }
<· undefined
> findMax([1,2,6,4,2,11])
<· 11
>
```

4. **Inheritance** is another place where the explicit change of context can be used.

Comment below if you know more reason :)

What are the cases where we need to take care of context?

We may lose the context i.e getting an `undefined` value for `this` in

1. Nested Functions

```
> var obj = {

      f1: function () {

      },

      f2: function (cb) {
          cb();
      },

      exec: function () {
          this.f2(function () {
              this.f1();
          });
      }
  };
< undefined
> obj.exec()
▶ Uncaught TypeError: this.f1 is not a function
      at <anonymous>:13:18
      at Object.f2 (<anonymous>:8:9)
      at Object.exec (<anonymous>:12:14)
      at <anonymous>:1:5
>
```

We need to keep the context of the `obj` object referenced for when the callback function is called, in the above, that does not happen and we get the error.

We can get rid of the above error by replacing the `exec` code with below

```
// use of bind
exec: function () {
    this.f2(function () {
        this.f1();
    }.bind(this));
}

// use of arrow function
exec: function () {
```


```
// another way not recommended though
exec: function () {
   var that = this;
    this.f2(() => {
        that.f1();
    });
}
```

2. Method as callback

```
let obj = {
  name: "test",
  waveHi() {
    return ('Hi',this.name);
  }
};

setTimeout(obj.waveHi, 1000)
```

The above will return `Hi undefined`, think for a second why? This is because the last line will be turn out to be

```
let f = obj.waveHi;
setTimeout(f, 1000);
```

and, `setTimeout` got the function `obj.waveHi`, separately from the object `obj`

Solutions are

```
// Wrapping function
setTimeout(function() {
  obj.waveHi(); // Hi Test
}, 1000);

// Arrow function
setTimeout(() => obj.waveHi(), 1000); // Hi Test
```



> *Note:*
>
> *1. Creating a “bound method reference” requires an anonymous wrapper function, and a calling cost. In specific situations, leveraging closures may be a better alternative.*
>
> *2. Any sort of function reference (assigning as a value, passing as an argument) loses the function’s original binding.*

Please consider **entering your email here** if you’d like to be added to my email list and **follow me on medium to read more article on javascript and on github to see my crazy code**. If anything is not clear or you want to point out something, please comment down below.

You may also like my other articles

1. Javascript Execution Context and Hoisting
2. Javascript — Generator-Yield/Next & Async-Await 😉
3. Javascript data structure with map, reduce, filter
4. Javascript- Currying VS Partial Application
5. Javascript ES6 — Iterables and Iterators
6. Javascript — Proxy, Javascript — Scopes

If you liked the article, please feel free to share and help others find it!

THANK YOU!