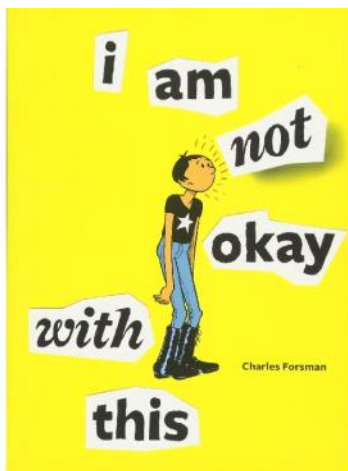


<WA1/>  
<AW1/>  
2021

## The 'this' keyword

"The" language of the Web

Fulvio Corno  
Luigi De Russis  
Enrico Masala



JavaScript: The Definitive Guide, 7th Edition  
Chapter 8. Classes

You Don't Know JS: this & Object Prototypes

JavaScript – The language of the Web

## 'THIS'

Meaning, usage, semantics

Common to all oop languages, but Javascript does things its own way

Applicazioni Web 1 - Web Applications 1 - 2020/2021

2

Usually (Java) this refers to the instance of the object in which it appears, but in JS it is not true!

## 'this' in JavaScript

- Given the peculiar treatment of Objects in JS, the 'this' keyword behaves differently than other OO languages
  - 'this' does not refer to the function in which it appears
  - 'this' does not (always) refer to the current object (functions are not always bound as object methods)
  - 'this' does not refer to the context (i.e., external function) in which the function is defined
  - 'this' does not refer to the object that generated the call (e.g., the object generating an event)
- Nevertheless, 'this' is extremely useful in callbacks and object methods
  - We must learn its rules...

method of an object can access this.attributes. No problem.  
BUT if I define an EXTERNAL function () => return this.age<18  
What does "this" REFERS TO? It is an external function, so it is not clear who that this is referring to!

The meaning of this is not related to WHERE does this appear (context)!

Applicazioni Web 1 - Web Applications 1 - 2020/2021

3

# The Golden Rule

- Within each function, the **'this'** keyword is always *bound* to some specific **object**
- The binding of **'this'** depends *exclusively on the call site* of the function (how the function is called)
  - ⚠ Does not depend on *how* the function is declared (function expression, function statement, passed reference, ...)
  - ⚠ Does not depend on *where* the function is declared (global, object property, nested, ...)
- 🚫 Notable exception: Arrow Functions (see at the end)

Binding of this to a specific object only depends on the call site of the function:  
IN WHICH PLACE OF THE CODE THE FUNCTION IS CALLED!

It is not true for arrow functions!

4

Applicazioni Web 1 - Web Applications 1 - 2020/2021

## The Call Site Of a Function

N.B. The same function could be called from different sites in the code, so we should look at the call stack to see at runtime in which of the places the function was called, to understand which object does this refers to

- Locate where the function is called from
  - Imagine being in a function, just called
  - Go back one step in the *call stack*, and check where you were just before being called
  - That location is the true call site
- The same function might be called from different places, in different times
  - Each time, the call site for *that invocation* is the *only* important information

5

Applicazioni Web 1 - Web Applications 1 - 2020/2021

## Sample Call Site Analysis

Try me!

```
function baz() {  
  // call-stack is: 'baz'  
  // so, our call-site is in the global scope  
  console.log( "baz" );  
  bar(); // <- call-site for 'bar'  
}  
  
function bar() {  
  // call-stack is: 'baz' -> 'bar'  
  // so, our call-site is in 'baz'  
  console.log( "bar" );  
  foo(); // <- call-site for 'foo'  
}  
  
function foo() {  
  // call-stack is: 'baz' -> 'bar' -> 'foo'  
  // so, our call-site is in 'bar'  
  console.log( "foo" );  
}  
  
baz(); // <- call-site for 'baz'
```

```
1 function baz() {  
2   // call-stack is: 'baz'  
3   // so, our call-site is in the global scope  
4   console.log( "baz" );  
5   bar(); // <- call-site for 'bar'  
6 }  
7  
8  
9 function bar() {  
10  // call-stack is: 'baz' -> 'bar'  
11  // so, our call-site is in 'baz'  
12  
13  console.log( "bar" );  
14  foo(); // <- call-site for 'foo'  
15 }  
16  
17 function foo() {  
18  // call-stack is: 'baz' -> 'bar' -> 'foo'  
19  // so, our call-site is in 'bar'  
20  
21  console.log( "foo" );  
22 }  
23  
24 baz(); // <- call-site for 'baz'
```

6

Applicazioni Web 1 - Web Applications 1 - 2020/2021

## Rule #1: Default Binding

- Standalone function invocation
    - `let a = foo();`
      - Normal function call
      - Default rule, applies if other special cases don't apply
  - When in strict mode, 'this' inside 'foo' is **undefined** (as expected!)
  - When not in strict mode, 'this' inside 'foo' is **the global object**
    - **global** in nodejs, or **window** in the browser
  - It is **useless**, no reason to use it (designed to be called)
- RULE: — ⚠ Never use 'this' inside functions called in standalone mode

## Rule #2: Implicit Binding

=automatic binding=dot notation=normal use

- Called in the context of an object (method)
  - `let a = obj.foo();` (method)
- foo is a (function-valued) property of obj
  - Defined inline with a function expression
  - Defined elsewhere but assigned to a property
- Inside foo(), **this** refers to obj
  - The specific object instance on which the function is called
  - `this.a` refers to property a of obj

```
function extrafoo() {
  console.log( this.a );
}

let obj = {
  a: 2,
  foo: extrafoo
};

obj.foo(); // 2
```

## Beware: Losing The Object Reference

```
function foo() {
  console.log( this.a );
}

let obj = {
  a: 2,
  foo: foo
};

let bar = obj.foo;
// function reference/alias!

bar(); // "oops, global"
```

Call Site

Reference to the object has been lost!!!! Lose implicit binding! Doesn't work!  
(bar is just a reference to foo[]), we don't have the information about from which object it was extracted ("obj")!

```
function foo() {
  console.log( this.a );
}

function doFoo(fn) {
  // `fn` is just a reference to `foo`
  fn();
}

let obj = {
  a: 2,
  foo: foo
};

doFoo( obj.foo ); // "oops, global"
```

Call Site

## Beware: Losing The Object Reference

```
function foo() {
  console.log( this.a );
}

let obj = {
  a: 2,
  foo: foo
};

let bar = obj.foo;
// function reference/a
bar(); // "oops, global"
```

```
function foo() {
  console.log( this.a );
}

function doFoo(fn) {
  // `fn` is just a reference to `foo`
  fn();
}
```

RULE: ALWAYS CALL "METHODS" THROUGH THE OBJECT ON WHICH WE WANT THEM TO BE CALLED

Must be careful, if we pass the function reference around, we lose the object reference, and the "default binding" will be applied.

👉 Always pass objects, never functions, if you want 'this' to work in the passed object 👈

10

## Rule #3: Explicit Binding

(Rarely used)    Used inside libraries to keep calls generic

- You may call a function indirectly, with a *calling method* (natively defined for all JS functions)

```
let y = foo.call(object, param, param, param) // (Methods of the function object)
let y = foo.apply(object, [param, param, param])
```
- In this case the call to foo is *explicitly bound* to the object (1<sup>st</sup> parameter)
  - Inside the function, this is bound to object
  - It basically behaves like object.foo(), even if foo is not a property of object.
- Often used inside libraries, rarely in the final programs

Applicazioni Web 1 - Web Applications 1 - 2020/2021

11

## Hard Binding

Special case of explicit binding

- Even the explicit binding may be "lost", if you pass the function around (instead of passing the object)
- You may force a binding to a function using its `.bind()` method to construct a new 'bound' function

```
let newfoo = foo.bind(object) // newfoo is a bound function
let y = newfoo(params)
```

Stores also reference to the object "obj" (that we lost before)

Const IsBobMinor = isOfMinorAge(bob); saves both the reference to the function and the reference to the object, but this is rarely useful since it will always be bound only to bob!  
Useful for passing properties around when using React.
- The newfoo function will always be bound to **object**

Applicazioni Web 1 - Web Applications 1 - 2020/2021

12

## Rule #4: new Binding

Function with new keyword

- When an object is created with a **constructor function** call, the function is bound to the newly created object

```
let obj = new Foo() ;
```

- Within Foo, **this** refers to the new object (later assigned to obj)

13

Applicazioni Web 1 - Web Applications 1 - 2020/2021

### Aside: How 'new' Works

- JS constructor call
  - when a function is invoked with new in front of it

```
let object = new Func() ;
```



- a brand-new object `{}` is created (aka, constructed) out of thin air
- the newly constructed object is `[[Prototype]]`-linked (*not relevant now*)
- the newly constructed object is set as the **this** binding for that function call
- unless the function returns its own alternate object, the new-invoked function call will automatically return the newly constructed object.

We bind the **this** keyword to the object that we are creating (inside the constructor all references to **this**, both in properties and methods, will refer to this object)

14

Applicazioni Web 1 - Web Applications 1 - 2020/2021

### Summary Of Rules

- Is the function called with new (**new binding**)? If so, this is the newly constructed object.  

```
var bar = new Foo() ;
```
- Is the function called with `call` or `apply` (**explicit binding**), even hidden inside a *bind hard binding*? If so, **this** is the explicitly specified object.  

```
var bar = foo.call( obj2 ) ;
```

`Call, apply, bind`
- Is the function called with a context (**implicit binding**), otherwise known as an owning or containing object? If so, **this** is *that* context object.  

```
var bar = obj1.foo() ;
```
- Otherwise (**default binding**). If in *strict mode*, **this** is undefined, otherwise **this** is the global object (`global` in node, `window` in browsers).  

```
var bar = foo()
```

15

Applicazioni Web 1 - Web Applications 1 - 2020/2021

## Exception : Arrow Functions =>

THE ARROW DOES NOT REDEFINE THIS!  
It has a closure over the existing this

- ⊗ The above rules **do not apply to Arrow Functions** !!!

1st rule  
`let fun = (n) => { this.a=n; }`

- Arrow functions adopt the 'this' binding **from the enclosing function scope** (or global scope)
  - Check the call site *of the enclosing function*!
- Extremely handy in event handlers and callbacks

```
function foo() {  
  setTimeout(() => {  
    // `this` here is lexically  
    // adopted from `foo()`  
    console.log( this.a );  
  },100);  
}  
  
var obj = {  
  a: 2  
};  
  
foo.call( obj ); // 2
```

## In Practice...

Rule	Example at call site	Suggestion
	<code>let foo = function(n) { this.a = n ; } ;</code>	
4. New binding	<code>let y = new foo(3) ;</code>	<b>Normal usage for object constructors</b>
3. Explicit binding	<code>let y = foo.call(obj, n) ; let newfoo = foo.bind(obj) ;</code>	Seldom used in user code, mostly in libraries
2. Implicit binding	<code>let y = obj.foo() ;</code>	<b>Normal usage for object methods</b>
1. Default binding	<code>let y = foo() ;</code>	Never use. Does not work in Strict mode.
<b>Exception:</b> Arrow Functions	<code>let foo = (n)=&gt;{ this.a = n ; } Uses surrounding scope (closure over this)</code>	Useful in callbacks (event handlers, async functions, ...)

## In Practice...

Rule	Example at call site	Suggestion
	<code>let foo = function(n) { this.a = n ; } ;</code>	
4. New binding	<code>let y = new foo(3) ;</code>	<b>Normal usage for object constructors</b>
3. Explicit binding	<del><code>let y = foo.call(obj, n) ; let newfoo = foo.bind(obj) ;</code></del>	<del>Seldom used in user code, mostly in libraries</del>
2. Implicit binding	<code>let y = obj.foo() ;</code>	<b>Normal usage for object methods</b>
1. Default binding	<del><code>let y = foo() ;</code></del>	<del>Never use. Does not work in Strict mode.</del>
<b>Exception:</b> Arrow Functions	<code>let foo = (n)=&gt;{ this.a = n ; } Uses surrounding scope (closure over this)</code>	Useful in callbacks (event handlers, async functions, ...)

N.B. If the call site is inside a library where our callback is called, we don't have any control over it!  
It is better to use the arrow function, to have maximum control, since it doesn't depend on the call site where the callback function will be called, but it depends on the enclosing function of our callback function. This is a shortcut to avoid the need to inspect inside the library how is structured the call site where our callback will be called

## References

- You Don't Know JS: this & Object Prototypes - 1st Edition, Kyle Simpson,  
<https://github.com/getify/You-Dont-Know-JS/tree/1st-ed/this%20%26%20Object%20prototypes> , Chapter 1 and Chapter 2

## References

- You Don't Know JS: this & Object Prototypes - 1st Edition, Kyle Simpson, <https://github.com/getify/You-Dont-Know-JS/tree/1st-ed/this%20%26%20Object%20prototypes> , Chapter 1 and Chapter 2



## License

- These slides are distributed under a Creative Commons license “**Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)**”
- **You are free to:**
  - **Share** — copy and redistribute the material in any medium or format
  - **Adapt** — remix, transform, and build upon the material
  - The licensor cannot revoke these freedoms as long as you follow the license terms.
- **Under the following terms:**
  - **Attribution** — You must give [appropriate credit](#), provide a link to the license, and [indicate if changes were made](#). You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - **NonCommercial** — You may not use the material for [commercial purposes](#).
  - **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the [same license](#) as the original.
  - **No additional restrictions** — You may not apply legal terms or [technological measures](#) that legally restrict others from doing anything the license permits.
- <https://creativecommons.org/licenses/by-nc-sa/4.0/>

