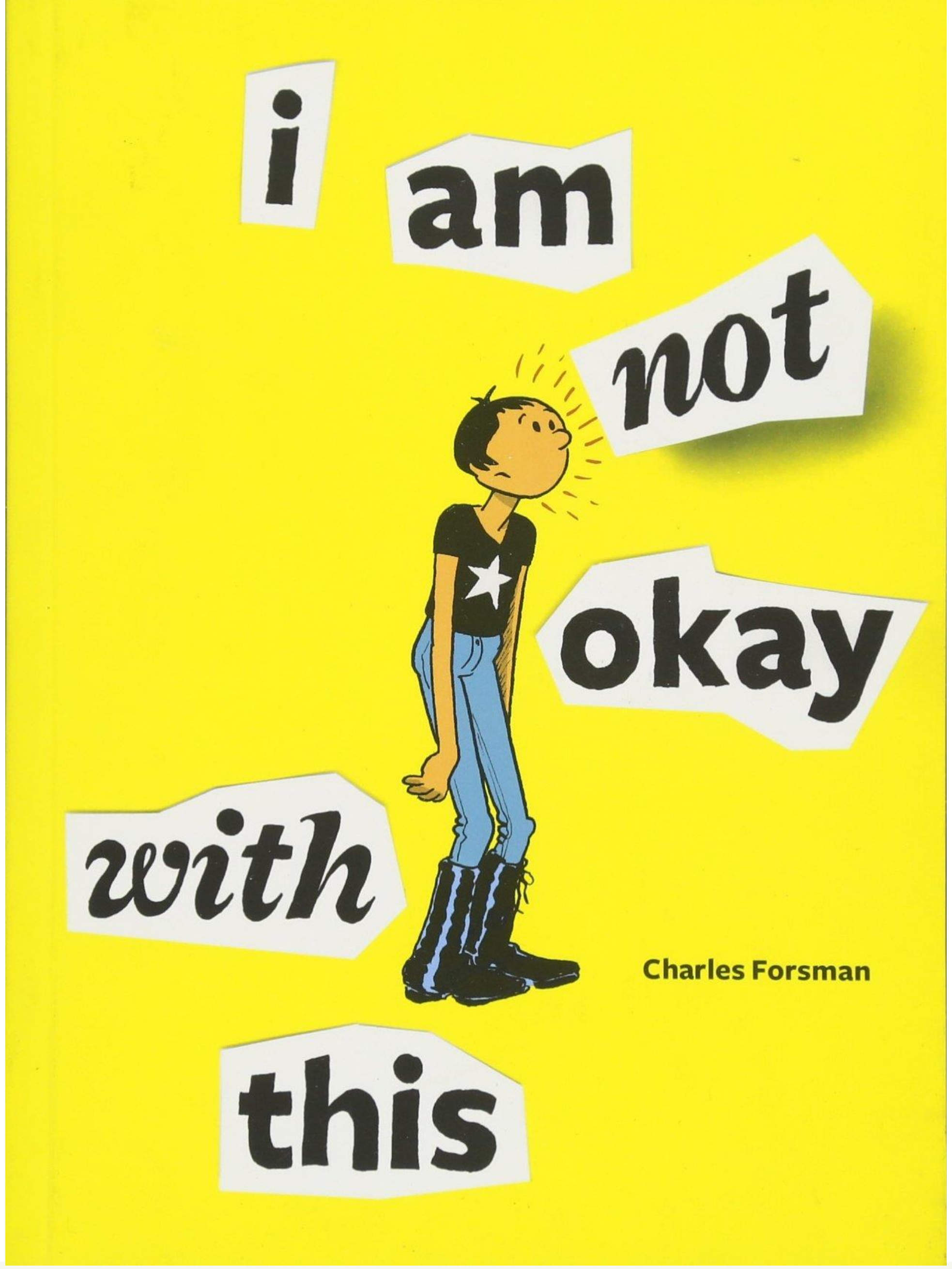
<WA/>
2025

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’

‘`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…

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)

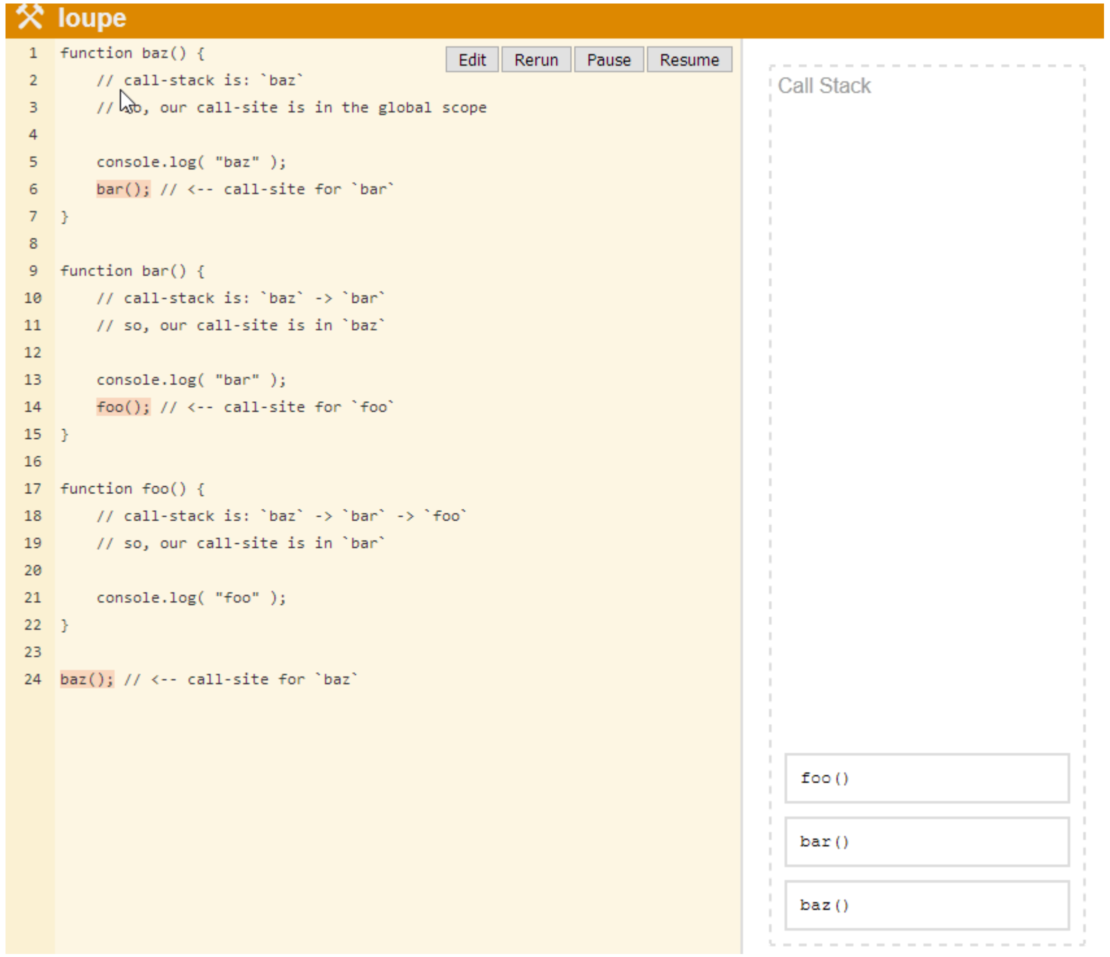
The *Call Site* Of a Function

- 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

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`
```



```
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`
```

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`**
- 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
  - Never use '`this`' inside functions called in standalone mode

Rule #2: Implicit Binding

- Called in the context of an object (method)

  `let a = obj.foo() ;`
- `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!

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

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

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

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

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

Beware: Losing The Object Reference

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

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

let bar = obj.foo;
// function reference/
```

Call Site

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

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

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

Call Site

```
/ "oops, global"
```

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 👁‍🗨

Rule #3: Explicit Binding

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

```
let y = foo.call(object, param, param, param)
let y = foo.apply(object, [param, param, param])
```

- In this case the call to `foo` is *explicitly bound* to the `object` (1st 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

Hard 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)
```

- The `newfoo` function will always be bound to `object`

Rule #4: `new` Binding

- 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)

Aside: How ‘**`new`**’ Works

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

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

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

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 ) ;
```

- 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()
```

Exception : Arrow Functions =>

- 🚫The above rules **do not apply** to Arrow Functions

  `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
	`let foo = function(n) { this.a = n ; }`	
4. New binding	`let y = new Foo(3) ;`	**Normal usage for object constructors**

3. Explicit binding	`let y = foo.call(obj, n) ;`
	`let newfoo = foo.bind(obj) ;`
		Seldom used in user code, mostly in libraries

2. Implicit binding	`let y = obj.foo() ;`	**Normal usage for object methods**
1. Default binding	`let y = foo() ;`	Never use. Does not work in Strict mode.
Exception: Arrow Functions	`let foo = (n)=>{ this.a = n ; }` Uses surrounding scope (closure over `this`)	Useful in callbacks (event handlers, async functions, …)

In Practice…

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

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/