
THIS

WHAT IS GOING ON HERE?

```
export default class Fabrice extends PureComponent {
  constructor() {
    this.onKillSheep = this.onKillSheep.bind(this);
  }

  onKillSheep() {
    ...
```

3 COMMON ASSUMPTIONS

```
var hi = function() {
  this.a = "'Ello";
};
```

- `this` refers to the function itself - no.
- `this` refers to the function's scope - nope.

4THIS

- `this` allows you to implicitly pass around different objects' contexts
- you could do this manually...

5 MANUALLY

```
function identify(context) {
    return context.name.toUpperCase();
}

function speak(context) {
    var greeting = "Hello, I'm " + identify( context );
    console.log( greeting );
}

var me = {
    name: "Jack"
};

var you = {
    name: "Nicolas"
};
```

6 THEORY

- `this` is a *run-time*, not *author-time*, binding
- it is contextually based on the conditions of the function's *invocation* (how it is called), not how it is declared.

7 ENGINE AND SH*T

- when a function is invoked, an *activation record* (a.k.a. an *execution context*) is created.
- this contains info about:
  - where the function was called from (call stack)
  - how the function was invoked
  - what parameters were passed
  - the `this` reference (used for the duration of the function's execution)

8 CALL SITE

- to decipher what `this` refers too, we must understand the call-site - where it was called
- look at the call stack (the stack of functions that have been called to get us to the current moment in execution)
- The call-site we care about is in the invocation before the currently executing function (i.e. *where* it was called)

9 CALL STACK

```
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`
}
```

10 THIS BINDING RULES

1. Default binding
2. Implicit binding
3. Explicit binding
4. new binding

11 DEFAULT BINDING

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

var a = 2;

foo(); // 2
```

- variables declared in the global scope are synonymous with global-object properties of the same name
- default binding - `this` points at the global object

12 IMPLICIT BINDING

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

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

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

- the call-site uses the `obj` context to reference the function, so you could say that the `obj` object "owns" or "contains" the function reference at the time the function is called.

13 EXPLICIT BINDING

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

var obj = {
    a: 2
};

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

- invoking foo with explicit binding by `foo.call(..)` allows us to force its `this` to be `obj`

14 EXPLICIT 2 - HARD BINDING

```
function foo(something) {
    console.log( this.a, something );
    return this.a + something;
}

var obj = {
    a: 2
};

var bar = foo.bind( obj );

var b = bar( 3 ); // 2 3
console.log( b ); // 5
```

- `bind(..)` returns a new function that is hard-coded to call the original function with the `this` context set as you specified.

15 NEW BINDING

```
function foo(a) {
    this.a = a;
}

var bar = new foo( 2 );
console.log( bar.a ); // 2
```

- new is not class based OOP

16 NEW BINDING

- what does new do?
  1. a brand new object is created (aka, constructed) out of thin air
  2. the newly constructed object is [[Prototype]]-linked
  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.

17 PRECEDENCE

- default < implicit < explicit < hard < new