You Don't Know JS Yet: Objects & Classes - 2nd Edition

Chapter 1: `this` Or That?

NOTE: |
:— |
Work in progress |

.

.

.

.

.

.

.

NOTE: |
:— |
Everything below here is previous text from 1st edition, and is only here for reference while 2nd edition work is underway. **Please ignore this stuff.** |

One of the most confused mechanisms in JavaScript is the `this` keyword. It's a special identifier keyword that's automatically defined in the scope of every function, but what exactly it refers to bedevils even seasoned JavaScript developers.

> Any sufficiently *advanced* technology is indistinguishable from magic.
> – Arthur C. Clarke

JavaScript's `this` mechanism isn't actually *that* advanced, but developers often paraphrase that quote in their own mind by inserting "complex" or "confusing", and there's no question that without lack of clear understanding, `this` can seem downright magical in *your* confusion.

Note: The word "this" is a terribly common pronoun in general discourse. So, it can be very difficult, especially verbally, to determine whether we are using "this" as a pronoun or using it to refer to the actual keyword identifier. For clarity, I will always use `this` to refer to the special keyword, and "this" or *this* or this otherwise.

Why `this`?

If the `this` mechanism is so confusing, even to seasoned JavaScript developers, one may wonder why it's even useful? Is it more trouble than it's worth? Before we jump into the *how*, we should examine the *why*.

Let's try to illustrate the motivation and utility of `this`:

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

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

var me = {
    name: "Kyle"
};

var you = {
    name: "Reader"
};

identify.call( me ); // KYLE
identify.call( you ); // READER

speak.call( me ); // Hello, I'm KYLE
speak.call( you ); // Hello, I'm READER
```

If the *how* of this snippet confuses you, don't worry! We'll get to that shortly. Just set those questions aside briefly so we can look into the *why* more clearly.

This code snippet allows the `identify()` and `speak()` functions to be re-used against multiple *context* (`me` and `you`) objects, rather than needing a separate version of the function for each object.

Instead of relying on `this`, you could have explicitly passed in a context object to both `identify()` and `speak()`.

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

function speak(context) {
```

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

identify( you ); // READER
speak( me ); // Hello, I'm KYLE
```

However, the `this` mechanism provides a more elegant way of implicitly "passing along" an object reference, leading to cleaner API design and easier re-use.

The more complex your usage pattern is, the more clearly you'll see that passing context around as an explicit parameter is often messier than passing around a `this` context. When we explore objects and prototypes, you will see the helpfulness of a collection of functions being able to automatically reference the proper context object.

Confusions

We'll soon begin to explain how `this` *actually* works, but first we must dispel some misconceptions about how it *doesn't* actually work.

The name "this" creates confusion when developers try to think about it too literally. There are two meanings often assumed, but both are incorrect.

Itself

The first common temptation is to assume `this` refers to the function itself. That's a reasonable grammatical inference, at least.

Why would you want to refer to a function from inside itself? The most common reasons would be things like recursion (calling a function from inside itself) or having an event handler that can unbind itself when it's first called.

Developers new to JS's mechanisms often think that referencing the function as an object (all functions in JavaScript are objects!) lets you store *state* (values in properties) between function calls. While this is certainly possible and has some limited uses, the rest of the book will expound on many other patterns for *better* places to store state besides the function object.

But for just a moment, we'll explore that pattern, to illustrate how `this` doesn't let a function get a reference to itself like we might have assumed.

Consider the following code, where we attempt to track how many times a function (`foo`) was called:

```
function foo(num) {
    console.log( "foo: " + num );
```

```
    // keep track of how many times `foo` is called
    this.count++;
}

foo.count = 0;

var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        foo( i );
    }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// how many times was `foo` called?
console.log( foo.count ); // 0 -- WTF?
```

`foo.count` is *still* `0`, even though the four `console.log` statements clearly indicate `foo(..)` was in fact called four times. The frustration stems from a *too literal* interpretation of what `this` (in `this.count++`) means.

When the code executes `foo.count = 0`, indeed it's adding a property `count` to the function object `foo`. But for the `this.count` reference inside of the function, `this` is not in fact pointing *at all* to that function object, and so even though the property names are the same, the root objects are different, and confusion ensues.

Note: A responsible developer *should* ask at this point, "If I was incrementing a `count` property but it wasn't the one I expected, which `count` *was* I incrementing?" In fact, were she to dig deeper, she would find that she had accidentally created a global variable `count` (see Chapter 2 for *how* that happened!), and it currently has the value `NaN`. Of course, once she identifies this peculiar outcome, she then has a whole other set of questions: "How was it global, and why did it end up `NaN` instead of some proper count value?" (see Chapter 2).

Instead of stopping at this point and digging into why the `this` reference doesn't seem to be behaving as *expected*, and answering those tough but important questions, many developers simply avoid the issue altogether, and hack toward some other solution, such as creating another object to hold the `count` property:

```
function foo(num) {
    console.log( "foo: " + num );
```

```
    // keep track of how many times ‘foo‘ is called
    data.count++;
}

var data = {
    count: 0
};

var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        foo( i );
    }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// how many times was ‘foo‘ called?
console.log( data.count ); // 4
```

While it is true that this approach "solves" the problem, unfortunately it simply ignores the real problem – lack of understanding what `this` means and how it works – and instead falls back to the comfort zone of a more familiar mechanism: lexical scope.

Note: Lexical scope is a perfectly fine and useful mechanism; I am not belittling the use of it, by any means (see *"Scope & Closures"* title of this book series). But constantly *guessing* at how to use `this`, and usually being *wrong*, is not a good reason to retreat back to lexical scope and never learn *why* `this` eludes you.

To reference a function object from inside itself, `this` by itself will typically be insufficient. You generally need a reference to the function object via a lexical identifier (variable) that points at it.

Consider these two functions:

```
function foo() {
    foo.count = 4; // ‘foo‘ refers to itself
}

setTimeout( function(){
    // anonymous function (no name), cannot
```

```
    // refer to itself
}, 10 );
```

In the first function, called a "named function", `foo` is a reference that can be used to refer to the function from inside itself.

But in the second example, the function callback passed to `setTimeout(..)` has no name identifier (so called an "anonymous function"), so there's no proper way to refer to the function object itself.

Note: The old-school but now deprecated and frowned-upon `arguments.callee` reference inside a function *also* points to the function object of the currently executing function. This reference is typically the only way to access an anonymous function's object from inside itself. The best approach, however, is to avoid the use of anonymous functions altogether, at least for those which require a self-reference, and instead use a named function (expression). `arguments.callee` is deprecated and should not be used.

So another solution to our running example would have been to use the `foo` identifier as a function object reference in each place, and not use `this` at all, which *works*:

```
function foo(num) {
    console.log( "foo: " + num );

    // keep track of how many times `foo` is called
    foo.count++;
}

foo.count = 0;

var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        foo( i );
    }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// how many times was `foo` called?
console.log( foo.count ); // 4
```

However, that approach similarly side-steps *actual* understanding of `this` and relies entirely on the lexical scoping of variable `foo`.

Yet another way of approaching the issue is to force `this` to actually point at the `foo` function object:

```
function foo(num) {
    console.log( "foo: " + num );

    // keep track of how many times ‘foo‘ is called
    // Note: ‘this‘ IS actually ‘foo‘ now, based on
    // how ‘foo‘ is called (see below)
    this.count++;
}

foo.count = 0;

var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        // using ‘call(..)‘, we ensure the ‘this‘
        // points at the function object (‘foo‘) itself
        foo.call( foo, i );
    }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// how many times was ‘foo‘ called?
console.log( foo.count ); // 4
```

Instead of avoiding `this`, we embrace it. We'll explain in a little bit *how* such techniques work much more completely, so don't worry if you're still a bit confused!

Its Scope

The next most common misconception about the meaning of `this` is that it somehow refers to the function's scope. It's a tricky question, because in one sense there is some truth, but in the other sense, it's quite misguided.

To be clear, `this` does not, in any way, refer to a function's **lexical scope**. It is true that internally, scope is kind of like an object with properties for each of the available identifiers. But the scope "object" is not accessible to JavaScript code. It's an inner part of the *Engine*'s implementation.

Consider code which attempts (and fails!) to cross over the boundary and use `this` to implicitly refer to a function's lexical scope:

```
function foo() {
    var a = 2;
    this.bar();
}

function bar() {
    console.log( this.a );
}

foo(); //undefined
```

There's more than one mistake in this snippet. While it may seem contrived, the code you see is a distillation of actual real-world code that has been exchanged in public community help forums. It's a wonderful (if not sad) illustration of just how misguided `this` assumptions can be.

Firstly, an attempt is made to reference the `bar()` function via `this.bar()`. It is almost certainly an *accident* that it works, but we'll explain the *how* of that shortly. The most natural way to have invoked `bar()` would have been to omit the leading `this.` and just make a lexical reference to the identifier.

However, the developer who writes such code is attempting to use `this` to create a bridge between the lexical scopes of `foo()` and `bar()`, so that `bar()` has access to the variable `a` in the inner scope of `foo()`. **No such bridge is possible.** You cannot use a `this` reference to look something up in a lexical scope. It is not possible.

Every time you feel yourself trying to mix lexical scope look-ups with `this`, remind yourself: *there is no bridge*.

What's `this`?

Having set aside various incorrect assumptions, let us now turn our attention to how the `this` mechanism really works.

We said earlier that `this` is not an author-time binding but a runtime binding. It is contextual based on the conditions of the function's invocation. `this` binding has nothing to do with where a function is declared, but has instead everything to do with the manner in which the function is called.

When a function is invoked, an activation record, otherwise known as an execution context, is created. This record contains information about where the function was called from (the call-stack), *how* the function was invoked, what parameters

were passed, etc. One of the properties of this record is the `this` reference which will be used for the duration of that function's execution.

In the next chapter, we will learn to find a function's **call-site** to determine how its execution will bind `this`.

Review (TL;DR)

`this` binding is a constant source of confusion for the JavaScript developer who does not take the time to learn how the mechanism actually works. Guesses, trial-and-error, and blind copy-n-paste from Stack Overflow answers is not an effective or proper way to leverage *this* important `this` mechanism.

To learn `this`, you first have to learn what `this` is *not*, despite any assumptions or misconceptions that may lead you down those paths. `this` is neither a reference to the function itself, nor is it a reference to the function's *lexical* scope.

`this` is actually a binding that is made when a function is invoked, and *what* it references is determined entirely by the call-site where the function is called.