You Don't Know JS: *this* & Object Prototypes

Chapter 2: `this` All Makes Sense Now!

In Chapter 1, we discarded various misconceptions about `this` and learned instead that `this` is a binding made for each function invocation, based entirely on its **call-site** (how the function is called).

Call-site

To understand `this` binding, we have to understand the call-site: the location in code where a function is called (**not where it's declared**). We must inspect the call-site to answer the question: what's *this* `this` a reference to?

Finding the call-site is generally: "go locate where a function is called from", but it's not always that easy, as certain coding patterns can obscure the *true* call-site.

What's important is to think about 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.

Let's demonstrate call-stack and call-site:

```
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`
```

Take care when analyzing code to find the actual call-site (from the call-stack), because it's the only thing that matters for `this` binding.

Note: You can visualize a call-stack in your mind by looking at the chain of function calls in order, as we did with the comments in the above snippet. But this is painstaking and error-prone. Another way of seeing the call-stack is using a debugger tool in your browser. Most modern desktop browsers have built-in developer tools, which includes a JS debugger. In the above snippet, you could have set a breakpoint in the tools for the first line of the `foo()` function, or simply inserted the `debugger;` statement on that first line. When you run the page, the debugger will pause at this location, and will show you a list of the functions that have been called to get to that line, which will be your call stack. So, if you're trying to diagnose `this` binding, use the developer tools to get the call-stack, then find the second item from the top, and that will show you the real call-site.

Nothing But Rules

We turn our attention now to *how* the call-site determines where `this` will point during the execution of a function.

You must inspect the call-site and determine which of 4 rules applies. We will first explain each of these 4 rules independently, and then we will illustrate their order of precedence, if multiple rules *could* apply to the call-site.

Default Binding

The first rule we will examine comes from the most common case of function calls: standalone function invocation. Think of *this* `this` rule as the default catch-all rule when none of the other rules apply.

Consider this code:

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

var a = 2;

foo(); // 2
```

The first thing to note, if you were not already aware, is that variables declared in the global scope, as `var a = 2` is, are synonymous with global-object properties of the same name. They're not copies of each other, they *are* each other. Think of it as two sides of the same coin.

Secondly, we see that when `foo()` is called, `this.a` resolves to our global variable `a`. Why? Because in this case, the *default binding* for `this` applies to the function call, and so points `this` at the global object.

How do we know that the *default binding* rule applies here? We examine the call-site to see how `foo()` is called. In our snippet, `foo()` is called with a plain, un-decorated function reference. None of the other rules we will demonstrate will apply here, so the *default binding* applies instead.

If `strict mode` is in effect, the global object is not eligible for the *default binding*, so the `this` is instead set to `undefined`.

```
function foo() {
        "use strict";

        console.log( this.a );
}

var a = 2;

foo(); // TypeError: `this` is `undefined`
```

A subtle but important detail is: even though the overall `this` binding rules are entirely based on the call-site, the global object is **only** eligible for the *default binding* if the **contents** of `foo()` are **not** running in `strict mode`; the `strict mode` state of the call-site of `foo()` is irrelevant.

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

var a = 2;

(function(){
        "use strict";

        foo(); // 2
})();
```

Note: Intentionally mixing `strict mode` and non-`strict mode` together in your own code is generally frowned upon. Your 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.

Implicit Binding

Another rule to consider is: does the call-site have a context object, also referred to as an owning or containing object, though *these* alternate terms could be slightly misleading.

Consider:

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

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

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

Firstly, notice the manner in which `foo()` is declared and then later added as a reference property onto `obj`. Regardless of whether `foo()` is initially declared *on* `obj`, or is added as a reference later (as this snippet shows), in neither case is the **function** really "owned" or "contained" by the `obj` object.

However, 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.

Whatever you choose to call this pattern, at the point that `foo()` is called, it's preceded by an object reference to `obj`. When there is a context object for a function reference, the *implicit binding* rule says that it's *that* object which should be used for the function call's `this` binding.

Because `obj` is the `this` for the `foo()` call, `this.a` is synonymous with `obj.a`.

Only the top/last level of an object property reference chain matters to the call-site. For instance:

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

var obj2 = {
        a: 42,
        foo: foo
};

var obj1 = {
        a: 2,
        obj2: obj2
};

obj1.obj2.foo(); // 42
```

Implicitly Lost

One of the most common frustrations that `this` binding creates is when an *implicitly bound* function loses that binding, which usually means it falls back to the *default binding*, of either the global object or `undefined`, depending on `strict mode`.

Consider:

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

var obj = {
        a: 2,
        foo: foo
```

```
};

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

var a = "oops, global"; // `a` also property on global object

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

Even though `bar` appears to be a reference to `obj.foo`, in fact, it's really just another reference to `foo` itself. Moreover, the call-site is what matters, and the call-site is `bar()`, which is a plain, un-decorated call and thus the *default binding* applies.

The more subtle, more common, and more unexpected way this occurs is when we consider passing a callback function:

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

function doFoo(fn) {
        // `fn` is just another reference to `foo`

        fn(); // <-- call-site!
}

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

var a = "oops, global"; // `a` also property on global object

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

Parameter passing is just an implicit assignment, and since we're passing a function, it's an implicit reference assignment, so the end result is the same as the previous snippet.

What if the function you're passing your callback to is not your own, but built-in to the language? No difference, same outcome.

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

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

var a = "oops, global"; // `a` also property on global object

setTimeout( obj.foo, 100 ); // "oops, global"
```

Think about this crude theoretical pseudo-implementation of `setTimeout()` provided as a built-in from the JavaScript environment:

```
function setTimeout(fn,delay) {
        // wait (somehow) for `delay` milliseconds
        fn(); // <-- call-site!
}
```

It's quite common that our function callbacks *lose* their `this` binding, as we've just seen. But another way that `this` can surprise us is when the function we've passed our callback to intentionally changes the `this` for the call. Event handlers in popular JavaScript libraries are quite fond of forcing your callback to have a `this` which points to, for instance, the DOM element that triggered the event. While that may sometimes be useful, other times it can be downright infuriating. Unfortunately, these tools rarely let you choose.

Either way the `this` is changed unexpectedly, you are not really in control of how your callback function reference will be executed, so you have no way (yet) of controlling the call-site to give your intended binding. We'll see shortly a way of "fixing" that problem by *fixing* the `this`.

Explicit Binding

With *implicit binding* as we just saw, we had to mutate the object in question to include a reference on itself to the function, and use this property function reference to indirectly (implicitly) bind `this` to the object.

But, what if you want to force a function call to use a particular object for the `this` binding, without putting a property function reference on the object?

"All" functions in the language have some utilities available to them (via their `[[Prototype]]` -- more on that later) which can be useful for this task. Specifically, functions have `call(..)` and `apply(..)` methods. Technically, JavaScript host environments sometimes provide functions which are special enough (a kind way of putting it!) that they do not have such functionality. But those are few. The vast majority of functions provided, and certainly all functions you will create, do have access to `call(..)` and `apply(..)`.

How do these utilities work? They both take, as their first parameter, an object to use for the `this`, and then invoke the function with that `this` specified. Since you are directly stating what you want the `this` to be, we call it *explicit binding*.

Consider:

```
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`.

If you pass a simple primitive value (of type `string`, `boolean`, or `number`) as the `this` binding, the primitive value is wrapped in its object-form (`new String(..)`, `new Boolean(..)`, or `new Number(..)`, respectively). This is often referred to as "boxing".

Note: With respect to `this` binding, `call(..)` and `apply(..)` are identical. They *do* behave differently with their additional parameters, but that's not something we care about presently.

Unfortunately, *explicit binding* alone still doesn't offer any solution to the issue mentioned previously, of a function "losing" its intended `this` binding, or just having it paved over by a framework, etc.

Hard Binding

But a variation pattern around *explicit binding* actually does the trick. Consider:

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

var obj = {
        a: 2
};

var bar = function() {
        foo.call( obj );
};

bar(); // 2
setTimeout( bar, 100 ); // 2

// `bar` hard binds `foo`'s `this` to `obj`
// so that it cannot be overriden
bar.call( window ); // 2
```

Let's examine how this variation works. We create a function `bar()` which, internally, manually calls `foo.call(obj)`, thereby forcibly invoking `foo` with `obj` binding for `this`. No matter how you later invoke the function `bar`, it will always manually invoke `foo` with `obj`. This binding is both explicit and strong, so we call it *hard binding*.

The most typical way to wrap a function with a *hard binding* creates a pass-thru of any arguments passed and any return value received:

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

var obj = {
        a: 2
};

var bar = function() {
        return foo.apply( obj, arguments );
};

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

Another way to express this pattern is to create a re-usable helper:

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

// simple `bind` helper
function bind(fn, obj) {
        return function() {
                return fn.apply( obj, arguments );
        };
}

var obj = {
        a: 2
};

var bar = bind( foo, obj );

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

Since *hard binding* is such a common pattern, it's provided with a built-in utility as of ES5: `Function.prototype.bind`, and it's used like this:

```
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.

Note: As of ES6, the hard-bound function produced by `bind(..)` has a `.name` property that derives from the original *target function*. For example: `bar = foo.bind(..)` should have a `bar.name` value of `"bound foo"`, which is the function call name that should show up in a stack trace.

API Call "Contexts"

Many libraries' functions, and indeed many new built-in functions in the JavaScript language and host environment, provide an optional parameter, usually called "context", which is designed as a work-around for you not having to use `bind(..)` to ensure your callback function uses a particular `this`.

For instance:

```
function foo(el) {
        console.log( el, this.id );
}

var obj = {
        id: "awesome"
};

// use `obj` as `this` for `foo(..)` calls
[1, 2, 3].forEach( foo, obj ); // 1 awesome  2 awesome  3 awesome
```

Internally, these various functions almost certainly use *explicit binding* via `call(..)` or `apply(..)`, saving you the trouble.

`new` Binding

The fourth and final rule for `this` binding requires us to re-think a very common misconception about functions and objects in JavaScript.

In traditional class-oriented languages, "constructors" are special methods attached to classes, that when the class is instantiated with a `new` operator, the constructor of that class is called. This usually looks something like:

```
something = new MyClass(..);
```

JavaScript has a `new` operator, and the code pattern to use it looks basically identical to what we see in those class-oriented languages; most developers assume that JavaScript's mechanism is doing something similar. However, there really is *no connection* to class-oriented functionality implied by `new` usage in JS.

First, let's re-define what a "constructor" in JavaScript is. In JS, constructors are **just functions** that happen to be called with the `new` operator in front of them. They are not attached to classes, nor are they instantiating a class. They are not even special types of functions. They're just regular functions that are, in essence, hijacked by the use of `new` in their invocation.

For example, the `Number(..)` function acting as a constructor, quoting from the ES5.1 spec:

> 15.7.2 The Number Constructor
>
> When Number is called as part of a new expression it is a constructor: it initialises the newly created object.

So, pretty much any ol' function, including the built-in object functions like `Number(..)` (see Chapter 3) can be called with `new` in front of it, and that makes that function call a *constructor call*. This is an important but subtle distinction: there's really no such thing as "constructor functions", but rather construction calls *of* functions.

When a function is invoked with `new` in front of it, otherwise known as a constructor call, the following things are done automatically:

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.

Steps 1, 3, and 4 apply to our current discussion. We'll skip over step 2 for now and come back to it in Chapter 5.

Consider this code:

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

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

By calling `foo(..)` with `new` in front of it, we've constructed a new object and set that new object as the `this` for the call of `foo(..)`. **So `new` is the final way that a function call's `this` can be bound.** We'll call this *new binding*.

Everything In Order

So, now we've uncovered the 4 rules for binding `this` in function calls. *All* you need to do is find the call-site and inspect it to see which rule applies. But, what if the call-site has multiple eligible rules? There must be an order of precedence to these rules, and so we will next demonstrate what order to apply the rules.

It should be clear that the *default binding* is the lowest priority rule of the 4. So we'll just set that one aside.

Which is more precedent, *implicit binding* or *explicit binding*? Let's test it:

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

var obj1 = {
        a: 2,
        foo: foo
};

var obj2 = {
        a: 3,
        foo: foo
};

obj1.foo(); // 2
obj2.foo(); // 3

obj1.foo.call( obj2 ); // 3
obj2.foo.call( obj1 ); // 2
```

So, *explicit binding* takes precedence over *implicit binding*, which means you should ask **first** if *explicit binding* applies before checking for *implicit binding*.

Now, we just need to figure out where *new binding* fits in the precedence.

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

var obj1 = {
        foo: foo
};

var obj2 = {};

obj1.foo( 2 );
console.log( obj1.a ); // 2

obj1.foo.call( obj2, 3 );
console.log( obj2.a ); // 3

var bar = new obj1.foo( 4 );
console.log( obj1.a ); // 2
```

```
console.log( bar.a ); // 4
```

OK, *new binding* is more precedent than *implicit binding*. But do you think *new binding* is more or less precedent than *explicit binding*?

Note: `new` and `call` / `apply` cannot be used together, so `new foo.call(obj1)` is not allowed, to test *new binding* directly against *explicit binding*. But we can still use a *hard binding* to test the precedence of the two rules.

Before we explore that in a code listing, think back to how *hard binding* physically works, which is that `Function.prototype.bind(..)` creates a new wrapper function that is hard-coded to ignore its own `this` binding (whatever it may be), and use a manual one we provide.

By that reasoning, it would seem obvious to assume that *hard binding* (which is a form of *explicit binding*) is more precedent than *new binding*, and thus cannot be overridden with `new`.

Let's check:

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

var obj1 = {};

var bar = foo.bind( obj1 );
bar( 2 );
console.log( obj1.a ); // 2

var baz = new bar( 3 );
console.log( obj1.a ); // 2
console.log( baz.a ); // 3
```

Whoa! `bar` is hard-bound against `obj1`, but `new bar(3)` did **not** change `obj1.a` to be `3` as we would have expected. Instead, the *hard bound* (to `obj1`) call to `bar(..)` ***is*** able to be overridden with `new`. Since `new` was applied, we got the newly created object back, which we named `baz`, and we see in fact that `baz.a` has the value `3`.

This should be surprising if you go back to our "fake" bind helper:

```
function bind(fn, obj) {
        return function() {
                fn.apply( obj, arguments );
        };
}
```

If you reason about how the helper's code works, it does not have a way for a `new` operator call to override the hard-binding to `obj` as we just observed.

But the built-in `Function.prototype.bind(..)` as of ES5 is more sophisticated, quite a bit so in fact. Here is the (slightly reformatted) polyfill provided by the MDN page for `bind(..)`:

```
if (!Function.prototype.bind) {
        Function.prototype.bind = function(oThis) {
                if (typeof this !== "function") {
                        // closest thing possible to the ECMAScript 5
                        // internal IsCallable function
                        throw new TypeError( "Function.prototype.bind - what " +
                                "is trying to be bound is not callable"
                        );
                }

                var aArgs = Array.prototype.slice.call( arguments, 1 ),
                        fToBind = this,
                        fNOP = function(){},
                        fBound = function(){
                                return fToBind.apply(
                                        (
                                                this instanceof fNOP &&
                                                oThis ? this : oThis
```

```
                                        ),
                                        aArgs.concat( Array.prototype.slice.call( arguments ) )
                                    );
                            }
                    ;

                    fNOP.prototype = this.prototype;
                    fBound.prototype = new fNOP();

                    return fBound;
            };
}
```

Note: The `bind(..)` polyfill shown above differs from the built-in `bind(..)` in ES5 with respect to hard-bound functions that will be used with `new` (see below for why that's useful). Because the polyfill cannot create a function without a `.prototype` as the built-in utility does, there's some nuanced indirection to approximate the same behavior. Tread carefully if you plan to use `new` with a hard-bound function and you rely on this polyfill.

The part that's allowing `new` overriding is:

```
this instanceof fNOP &&
oThis ? this : oThis

// ... and:

fNOP.prototype = this.prototype;
fBound.prototype = new fNOP();
```

We won't actually dive into explaining how this trickery works (it's complicated and beyond our scope here), but essentially the utility determines whether or not the hard-bound function has been called with `new` (resulting in a newly constructed object being its `this`), and if so, it uses *that* newly created `this` rather than the previously specified *hard binding* for `this`.

Why is `new` being able to override *hard binding* useful?

The primary reason for this behavior is to create a function (that can be used with `new` for constructing objects) that essentially ignores the `this` *hard binding* but which presets some or all of the function's arguments. One of the capabilities of `bind(..)` is that any arguments passed after the first `this` binding argument are defaulted as standard arguments to the underlying function (technically called "partial application", which is a subset of "currying").

For example:

```
function foo(p1,p2) {
        this.val = p1 + p2;
}

// using `null` here because we don't care about
// the `this` hard-binding in this scenario, and
// it will be overridden by the `new` call anyway!
var bar = foo.bind( null, "p1" );

var baz = new bar( "p2" );

baz.val; // p1p2
```

Determining `this`

Now, we can summarize the rules for determining `this` from a function call's call-site, in their order of precedence. Ask these questions in this order, and stop when the first rule applies.

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

   `var bar = new foo()`

2. 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 )`

3. 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()`

4. Otherwise, default the `this` (**default binding**). If in `strict mode`, pick `undefined`, otherwise pick the `global` object.

   `var bar = foo()`

That's it. That's *all it takes* to understand the rules of `this` binding for normal function calls. Well... almost.

Binding Exceptions

As usual, there are some *exceptions* to the "rules".

The `this`-binding behavior can in some scenarios be surprising, where you intended a different binding but you end up with binding behavior from the *default binding* rule (see previous).

Ignored `this`

If you pass `null` or `undefined` as a `this` binding parameter to `call`, `apply`, or `bind`, those values are effectively ignored, and instead the *default binding* rule applies to the invocation.

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

var a = 2;

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

Why would you intentionally pass something like `null` for a `this` binding?

It's quite common to use `apply(..)` for spreading out arrays of values as parameters to a function call. Similarly, `bind(..)` can curry parameters (pre-set values), which can be very helpful.

```
function foo(a,b) {
        console.log( "a:" + a + ", b:" + b );
}

// spreading out array as parameters
foo.apply( null, [2, 3] ); // a:2, b:3

// currying with `bind(..)`
var bar = foo.bind( null, 2 );
bar( 3 ); // a:2, b:3
```

Both these utilities require a `this` binding for the first parameter. If the functions in question don't care about `this`, you need a placeholder value, and `null` might seem like a reasonable choice as shown in this snippet.

Note: We don't cover it in this book, but ES6 has the `...` spread operator which will let you syntactically "spread out" an array as parameters without needing `apply(..)`, such as `foo(...[1,2])`, which amounts to `foo(1,2)` -- syntactically avoiding a `this` binding if it's unnecessary. Unfortunately, there's no ES6 syntactic substitute for currying, so the `this` parameter of the `bind(..)` call still needs attention.

However, there's a slight hidden "danger" in always using `null` when you don't care about the `this` binding. If you ever use that against a function call (for instance, a third-party library function that you don't control), and that function *does* make a `this` reference, the *default binding* rule means it might inadvertently reference (or worse, mutate!) the `global` object (`window` in the browser).

Obviously, such a pitfall can lead to a variety of *very difficult* to diagnose/track-down bugs.

Safer `this`

Perhaps a somewhat "safer" practice is to pass a specifically set up object for `this` which is guaranteed not to be an object that can create problematic side effects in your program. Borrowing terminology from networking (and the military), we can create a "DMZ" (de-militarized zone) object -- nothing more special than a completely empty, non-delegated (see Chapters 5 and 6) object.

If we always pass a DMZ object for ignored `this` bindings we don't think we need to care about, we're sure any hidden/unexpected usage of `this` will be restricted to the empty object, which insulates our program's `global` object from side-effects.

Since this object is totally empty, I personally like to give it the variable name `ø` (the lowercase mathematical symbol for the empty set). On many keyboards (like US-layout on Mac), this symbol is easily typed with `⌥` + `o` (option+ `o`). Some systems also let you set up hotkeys for specific symbols. If you don't like the `ø` symbol, or your keyboard doesn't make that as easy to type, you can of course call it whatever you want.

Whatever you call it, the easiest way to set it up as **totally empty** is `Object.create(null)` (see Chapter 5). `Object.create(null)` is similar to `{ }`, but without the delegation to `Object.prototype`, so it's "more empty" than just `{ }`.

```
function foo(a,b) {
        console.log( "a:" + a + ", b:" + b );
}

// our DMZ empty object
var ø = Object.create( null );

// spreading out array as parameters
foo.apply( ø, [2, 3] ); // a:2, b:3

// currying with `bind(..)`
var bar = foo.bind( ø, 2 );
bar( 3 ); // a:2, b:3
```

Not only functionally "safer", there's a sort of stylistic benefit to `ø`, in that it semantically conveys "I want the `this` to be empty" a little more clearly than `null` might. But again, name your DMZ object whatever you prefer.

Indirection

Another thing to be aware of is you can (intentionally or not!) create "indirect references" to functions, and in those cases, when that function reference is invoked, the *default binding* rule also applies.

One of the most common ways that *indirect references* occur is from an assignment:

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

var a = 2;
var o = { a: 3, foo: foo };
var p = { a: 4 };

o.foo(); // 3
(p.foo = o.foo)(); // 2
```

The *result value* of the assignment expression `p.foo = o.foo` is a reference to just the underlying function object. As such, the effective call-site is just `foo()`, not `p.foo()` or `o.foo()` as you might expect. Per the rules above, the *default binding* rule applies.

Reminder: regardless of how you get to a function invocation using the *default binding* rule, the `strict mode` status of the **contents** of the invoked function making the `this` reference -- not the function call-site -- determines the *default binding* value: either the `global` object if in non-`strict mode` or `undefined` if in `strict mode`.

Softening Binding

We saw earlier that *hard binding* was one strategy for preventing a function call falling back to the *default binding* rule inadvertently, by forcing it to be bound to a specific `this` (unless you use `new` to override it!). The problem is, *hard-binding* greatly reduces the flexibility of a function, preventing manual `this` override with either the *implicit binding* or even subsequent *explicit binding* attempts.

It would be nice if there was a way to provide a different default for *default binding* (not `global` or `undefined`), while still leaving the function able to be manually `this` bound via *implicit binding* or *explicit binding* techniques.

We can construct a so-called *soft binding* utility which emulates our desired behavior.

```
if (!Function.prototype.softBind) {
        Function.prototype.softBind = function(obj) {
                var fn = this,
                        curried = [].slice.call( arguments, 1 ),
                        bound = function bound() {
                                return fn.apply(
                                        (!this ||
                                                (typeof window !== "undefined" &&
                                                        this === window) ||
                                                (typeof global !== "undefined" &&
                                                        this === global)
                                        ) ? obj : this,
                                        curried.concat.apply( curried, arguments )
                                );
                        };
                bound.prototype = Object.create( fn.prototype );
                return bound;
        };
}
```

The `softBind(..)` utility provided here works similarly to the built-in ES5 `bind(..)` utility, except with our *soft binding* behavior. It wraps the specified function in logic that checks the `this` at call-time and if it's `global` or `undefined`, uses a pre-specified alternate *default* (`obj`). Otherwise the `this` is left untouched. It also provides optional currying (see the `bind(..)` discussion earlier).

Let's demonstrate its usage:

```
function foo() {
   console.log("name: " + this.name);
}

var obj = { name: "obj" },
    obj2 = { name: "obj2" },
    obj3 = { name: "obj3" };

var fooOBJ = foo.softBind( obj );

fooOBJ(); // name: obj

obj2.foo = foo.softBind(obj);
obj2.foo(); // name: obj2   <---- look!!!

fooOBJ.call( obj3 ); // name: obj3   <---- look!

setTimeout( obj2.foo, 10 ); // name: obj   <---- falls back to soft-binding
```

The soft-bound version of the `foo()` function can be manually `this`-bound to `obj2` or `obj3` as shown, but it falls back to `obj` if the *default binding* would otherwise apply.

Lexical `this`

Normal functions abide by the 4 rules we just covered. But ES6 introduces a special kind of function that does not use these rules: arrow-function.

Arrow-functions are signified not by the `function` keyword, but by the `=>` so called "fat arrow" operator. Instead of using the four standard `this` rules, arrow-functions adopt the `this` binding from the enclosing (function or global) scope.

Let's illustrate arrow-function lexical scope:

```
function foo() {
        // return an arrow function
        return (a) => {
                // `this` here is lexically adopted from `foo()`
                console.log( this.a );
        };
}

var obj1 = {
        a: 2
};

var obj2 = {
        a: 3
};

var bar = foo.call( obj1 );
bar.call( obj2 ); // 2, not 3!
```

The arrow-function created in `foo()` lexically captures whatever `foo()` s `this` is at its call-time. Since `foo()` was `this` -bound to `obj1` , `bar` (a reference to the returned arrow-function) will also be `this` -bound to `obj1` . The lexical binding of an arrow-function cannot be overridden (even with `new` !).

The most common use-case will likely be in the use of callbacks, such as event handlers or timers:

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

var obj = {
        a: 2
};

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

While arrow-functions provide an alternative to using `bind(..)` on a function to ensure its `this` , which can seem attractive, it's important to note that they essentially are disabling the traditional `this` mechanism in favor of more widely-understood lexical scoping. Pre-ES6, we already have a fairly common pattern for doing so, which is basically almost indistinguishable from the spirit of ES6 arrow-functions:

```
function foo() {
        var self = this; // lexical capture of `this`
        setTimeout( function(){
                console.log( self.a );
        }, 100 );
}

var obj = {
        a: 2
};

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

While `self = this` and arrow-functions both seem like good "solutions" to not wanting to use `bind(..)` , they are essentially fleeing from `this` instead of understanding and embracing it.

If you find yourself writing `this` -style code, but most or all the time, you defeat the `this` mechanism with lexical `self = this` or arrow-function "tricks", perhaps you should either:

1. Use only lexical scope and forget the false pretense of `this` -style code.
2. Embrace `this` -style mechanisms completely, including using `bind(..)` where necessary, and try to avoid `self = this` and arrow-function "lexical this" tricks.

A program can effectively use both styles of code (lexical and `this`), but inside of the same function, and indeed for the same sorts of look-ups, mixing the two mechanisms is usually asking for harder-to-maintain code, and probably working too hard to be clever.

Review (TL;DR)

Determining the `this` binding for an executing function requires finding the direct call-site of that function. Once examined, four rules can be applied to the call-site, in *this* order of precedence:

1. Called with `new`? Use the newly constructed object.
2. Called with `call` or `apply` (or `bind`)? Use the specified object.
3. Called with a context object owning the call? Use that context object.
4. Default: `undefined` in `strict mode`, global object otherwise.

Be careful of accidental/unintentional invoking of the *default binding* rule. In cases where you want to "safely" ignore a `this` binding, a "DMZ" object like `ø = Object.create(null)` is a good placeholder value that protects the `global` object from unintended side-effects.

Instead of the four standard binding rules, ES6 arrow-functions use lexical scoping for `this` binding, which means they adopt the `this` binding (whatever it is) from its enclosing function call. They are essentially a syntactic replacement of `self = this` in pre-ES6 coding.