`this` or That

`this` is a special identifier keyword that's **automatically defined in the scope of every function**, but what exactly it refers to confuses even season JS developers

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

Why is keyword `this` necessary

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

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

let me = {
  name: "Kyle"
};

let 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
```

This code snippet allows the `identify()` and `speak()` functions to be resued against multiple *context* objects (`me` and `you`), 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) {
  let greeting = "Hello, I'm " + identify(context);
  console.log(greeting);
```

```
}

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

But rather than having to pass the `context` the `this` mechanism provides a more elegant way of implicitly "passing along" an object reference, leading to cleaner API design and easier reuse

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

Itself

The **first common temptation is to assume `this` refers to the function itself**...

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 JavaScript'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

We'll explore 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;

let 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 -- how is this possible...
```

foo.count is still 0, even though the 4 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..

But **if I was incrementing a `count` property but it wasn't the one I expected...**, which `count` *was* I incrementing? We **accidentally created a global variable:** `count` and this global variable `count` currently has the value `NaN`

But *How was it global? Why did it end up `NaN` instead of some proper count value?*

Some developers will find a "hack" that solves the problem **BUT** clearly show that they have no clear understanding of the problem or how `this` works Consider:

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

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

let data = {
  count: 0
};

let 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, it 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` **LEXICAL SCOPE IS GREAT**, but now when used to avoid the problem at hand

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 (called an "anonymous function"), so there's no proper way to refer to the function object itself

So another solution to our running example would have been to use the `foo` indentifer 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;

let 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` cal
console.log( foo.count ); // 4
```

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

But 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;

let 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
```

Above is the correct way of embracing `this` functionality

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

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

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

```
foo(); // ReferenceError: a is not defined
```

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

First, an attempt is made to reference the `bar()` function via `this.bar()` It is almost certainly an accident that it works, but we'll have to 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

Review

`this` binding is a constant source of confusion for the JavaScript developer who does not take the time to understand it

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** mechanism... that is made when **a function is invoked** and *what* it references is determined **entirely** by the **call-site where the function is called**

`this` All Makes Sense Now

We learned that `this` is a way of **binging** It binds 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 **call-site**: is the location in code where a function is **called** (**NOT** where it's declared)

We have to inspect the call-site to answer the question: 'what is *this* `this` a reference to?

What's important is to think about the *call-stack*... (the stack of functions that have been to get us tot he current moment in execution) The call-site we care about is *in* the invocation *before* the currently executing function

Let's demonstrate the 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 for `foo`

  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-suite (from the call-stack), because it's the only thing that matters for `this` binding

> 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 code snippet above But this is painstaking and error-prone Another way of seeing the call-stack is using a debugget tool in your browser In the previous 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

HOW the call-stie determines where `this` will point **during** the execution of a function

We have 4 different types of binding

1. Default Binding

Default binding is the most common case of function calls: **stand-alone function invocation** This of this `this` rule as the default catch-all rule when none of the other rules apply

Consider the following 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.

The are **NOT** copies of each other, **they are each other** Think of it as two sides of the same coin

The Second thing to note, 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 point `this` at the global object

How do we know that this is `default binding`? We examine the call-site to see how `foo()` is called In our snippet, `foo()` is called with a plain, undecorated 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);
}

let a = 2;

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

A subtle but improtant detail is that 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`;

2. Implicit Binding

Another rule to consider is whether the `call-site` has 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);
}

let obj = {
  a: 2,
  foo: foo // the function foo is added as a reference property onto the object: `obj`
};

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

First, 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* `foo`, or is added as a reference later, 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 preceeded 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 that 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);
}

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

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 eiher the global object or `undefined`, depending on `strict` mode Consider:

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

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

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

let 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, undecorated call, and this the *default binding* applies

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

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

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

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

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

let 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 into the language? No difference, same outcome:

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

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

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

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

// Think about this crude theoretical pseduoimplementation of
`setTimeout()`
// providede as bult-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 JS libraries are **quite fond** of forcing your callback to have a `this` that 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`

3. 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" function in the language have some utilities available to them, which can be useful for this task Specifically, functions have `call(..)` and `apply(..)` methods

Technically, JavaScript host environments sometimes provide functions that 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're directly stating what you want the `this` to be, we call it *explicit binding*

Consider:

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

let 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(..)`) This is referred to as "boxing"

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.

4. Hard Binding

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

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

let obj = {
  a:2
}

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

bar(); // 2
```

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

// hard-bound `bar` can no longer have its `this` overridden
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-through of any arguments passed and any return valued received:

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

let obj = {
  a:2
};

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

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

// Another way to express this pattern is to create a resuable 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);
  };
}

let obj = {
  a:2
};

let bar = bind(foo, obj);

let 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;
}

let obj = {
  a:2
}

let bar = foo.bind(obj);

let b = (3);
console.log(b);
```

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

API call "contexts"

Many libraries' functions, and indeed many new built-in function in the JavaScript language and host environment, provide an optional parameter, usually called "context", which is designed as a "work around" for you nor having to use `bind(..)` to ensure your callback function uses a particular `this`

For instance:

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

let 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 function almost certainly use *explicit binding* via `call(..)` or `apply(..)`, saving you the trouble.

`new` Binding

The final rule for `this` binding require us to rethink a very common misconception about functions and objects in JavaScript

In traditional class-oriented languages, "constructors" are special methods attached to classes, and 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.

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, consider the `Number(..)` function actiong as a constructor:

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

So, pretty much any function, including the built-in object functions like `Number(..)` 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 a `constuctor 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, thew `new`-invoked function call with *automatically* return thew newly construced object

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

Consider this code:

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

let 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 four rules for binding `this` in function calls. All you need to do is find the call-site and inspect it to see whichrule 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 four. So we'll just set that one aside.

Which is more precedent, *implicit binding* or *explicit binding*

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

let obj1 = {
  a:2,
  foo:food
};

let obj2 = {
  a:3,
  foo:food
};

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* aplies 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;
}
let obj1 = {
  foo: foo
};

let obj2 = {};

obj1.foo( 2 );

console.log( obj1.a ); // 2
obj1.foo.call( obj2, 3 );

console.log( obj2.a ); // 3
```

```
let 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*?

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

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

The primary reason ior this behavior is to create a function (that can be used with `new` for constructing objects) that essentially ignore 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 arguments are defaulted as standard arguments to the underlying function (technically called "partial application", which is a subset of "currying")

Consider:

```
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`

We can summarize the rules for determing `this` from a function'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 `let var = 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. `let bar = foo.call(obj2)`
3. Is the function called with a context *(implicit binding)*, otherwise known as owning or containing object? If so, `this` is *that* context object `let bar = obj1.foo()`
4. Otherwise, default the `this` *(default binding)*. If in `strict mode`, pick `undefined`, otherwise pick the `global` object. `let boo = foo()`

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

Lexical `this`

Normal functions abide by the four rules we just covered but ES6 introduces a special kind of function that does not use these rules: **the arrow function**

Arrow functions are signified not 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 the arrow-function lexical scope:

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

let obj1 = {
  a:2
}

let obj2 = {
  a:3
}

let 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 `this`-bound to `obj1` **SO** both functions `foo()` and `bar()` are 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 inherited from `foo()`
    console.log(this.a);
  }, 100);
}

let 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 Arrow functions are essentially are disabling the traditional `this` mechanism In favor of a more widely understood lexical scoping

Review pt. 2

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:

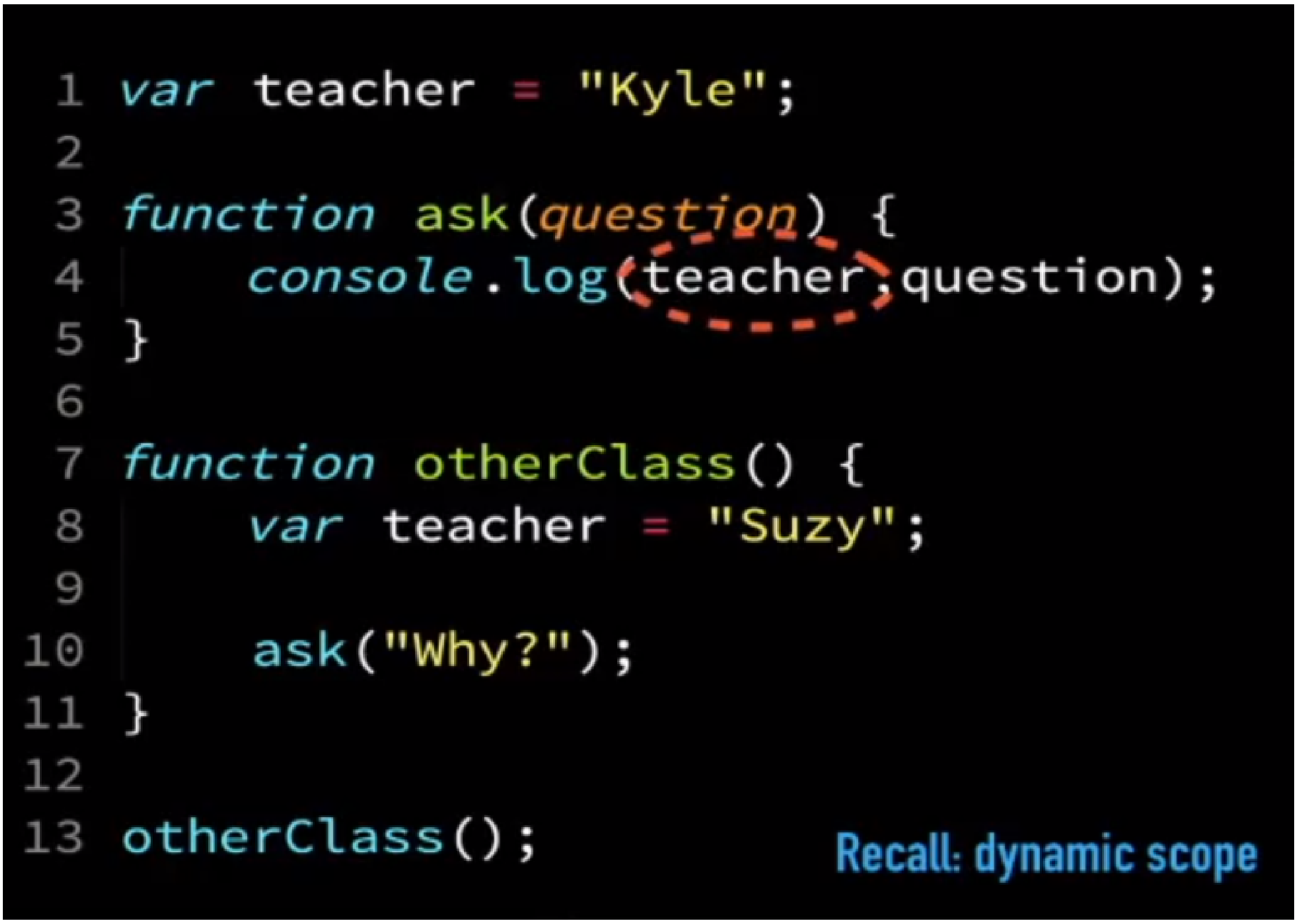
1. Called with `new`? Use the newly constructed object
2. Called with `call` or `apply` (or `bind`)? Use the specified object
3. Called with a context object owning the call? use that context object
4. Default: `undefined` in **strict mode**, global objject otherwise

Be carefulof 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 inherit the `this` binding (whatever it is) from encolising function acll They are essentially a syntactic replacement of `self = this` (which you would've seen in pre-ES6 coding)

this explained in the Video ############

Remember Dynamic Scoping:



Here we have a function that is **in a dynamic scope** So instead of asking `teacher`(on line 4) to get the teacher that's on line 1... it goes to line 8. Because `ask` was call from line 10, it was called from the other class scope... **that is what dynamic scope does**, And IN JS we have something similar But it doesnt depend on where something is, it depends on **how I call it**

A function's `this` references the **execution context for that call**, determined **entirely** by **how the function was called**

The defintion of the function **does not matter**, to determining the `this` keyword The only thing that matters is **how does that function get invoked**

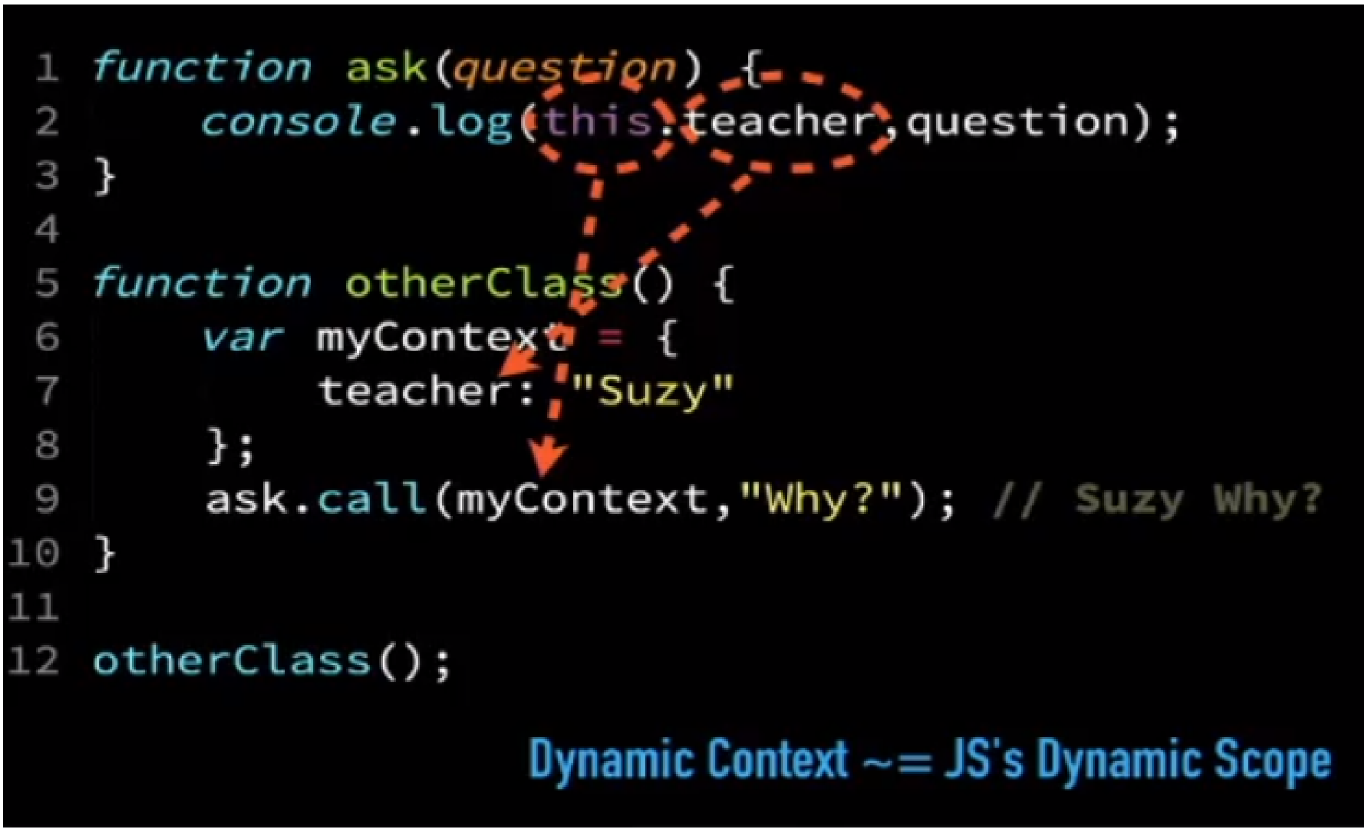
The `this` keyword is JavaScript's version of dynamic scope Its this way of having a flexible, reuable behavior

```
function ask(question) {
  console.log(this.teacher,question);
}

function otherClass() {
  var myContext = {
    teacher: "Suzy"
  };

  ask.call(myContext,"Why?") // Suzy Why?
}
```

```
otherClass();
```

So here we have a version of the `ask` function which is `this` aware (it uses a `this` keyword so it's `this`-aware) And you'll notice we're calling `ask` from some other location, **but that doesn't matter** It's not **where** I call from, **it's how I call it**

If I use `ask.call` on line 9, I am saying use "this particular object as your `this` keyword AND invoke the function in that context.



So the `this` keyword in this particular case, will end up pointing at `myContext`

So you see that sort of dynamic flexibility happening here You see that I could call that same `ask` function, lots of different ways And provide lots of different context objects for the `this` keyword to point on That's the dynamic flexible resuability of the `this` keyword

That's why it exists, it exists so that we can invoke functions in these different contexts

Summation

In Summation, `this` is entirely determined by how it is called, we tell it what context objects to use (to point to)

In the `ask` function we told it which context to point to! We could do many ways!

```
function ask(question) {
  console.log(this.teacher,question);
}

function otherClass() {
```

```
    var myContext = {
      teacher: "Suzy"
    };

    ask.call(myContext,"Why?") // Suzy Why?
  }

  functon otherClass2() {
    var myContext = {
      teacher: "Sharaby"
    };
    ask.call(myContext, "She was hot") // Sharaby She was hot
  }

  otherClass();
  otherClass2();
```

That's the **dynamic flexible resuability** of the `this` keyword

That's why it exists,**it exists so that we can invoke functions in these different contexts**

Explicit & Implicit Binding

Implicit

```
var workshop = {
  teacher: "kyle"
  ask(question) {
    console.log(this.teacher,questions)
  },
};

workshop.ask("What is implicit binding"); // kyle What is implicit binding
```

We have a `workshop` object that is `this`-aware When we get the `ask(question)` invoked, how does it figure out what the `this` keyword should point at? The answer is: **because of the call site**, because of the call site, the `this` keyword is gonna end up pointing at the object that is used to invoke it, which in this case is the line `workshop.ask("what is implicit binding")` `workshop.ask` says **invoke `ask` with the `this` keyword pointing at workshop** Thats what the implicit binding rule says This particular rule is the most common and intuitive (but it's only 1 of 4 ways)

implicit binding is how we share behavior among different contexts

```
function ask(question) {
  console.log(this.teacher,question);
}

var workshop1 = {
```

```
  teacher: "kyle",
  ask: ask, // reference to ask function
};

var workshio2 = {
  teacher: "suzy",
  ask: ask, // reference to ask function
};

workshop1.ask("How do I share a method");
// kyle How do I share a method

workshop2.ask("How do I share a method");
// suzy How do I share a method
```

Here I only define 1 ask function, **but** I'm sharing the ask function across 2 different objects: `workshop1` & `workshop2` 2 separate objects with 2 separate pieces of data in them **But**, because both objects have a reference to the ask function on it When I use that reference to invoke the ask function, the implicit binding rule says: **invoke that one funtion in a different context each time**!

So we don't have two ask functions there, just one, but it's invoked in 2 different contexts, and we can do this in an infinite amount of contexts

Explicit

```
function ask(question) {
  console.log(this.teacher,question);
}

var workshop1 = {
  teacher: "kyle",
};

var workshio2 = {
  teacher: "suzy",
};

workshop1.ask("Can I explicitly set context");
// kyle Can I explicitly set context

workshop2.ask("Can I explicitly set context");
// suzy Can I explicitly set context
```

There's another way to invoke functions, the `.call` method along with its cousin `.apply()` method, both of them take, as their first agument, a `this` keyword

So when we say: `workshop1.ask("Can I explicitly set context")` & `workshop2.ask("Can I explicitly set context")` this is saying invoke the ask function with the `this` context of `workshop1` and invoke the ask function with the `this` context of `workshop2` Wherever this function(`ask`) comes from, invoke it in a particular context which I'm going to specify **So we can use `.call` and `.apply` to explicitly tell JavaScript which context to invoke it in**!

And a subrule of the explicit binding is **losing your `this` binding** If you've ever worked with a function that you pass around, and all of a sudden it used to have a `this` binding and now it doesnt have a `this` binding It's very frustating when you think of a `this` keyword as being predictable and then you find out its not predictable, but flexible

So a variation of explicit binding is called **hard binding**

```
var workshop = {
  teacher: "kyle",
  ask(question) {
    console.log(this.teacher, question);
  },
};

setTimeout(workshop.ask,10,"Lost this?");
// undefined Lost this?

setTimeout(workshop.ask.bind(workshop),10,"hard bound this?");
// kyle Hard bound this?
```

Looking at this line: `setTimeout(workshop.ask,10,"Lost this?")`, if I passed in `workshop.ask`, that method **is on the workshop object**, but that line is not the call site

You have to imagine in your head, **what would the call site look like** for the function whenever that timer ran 10 seconds from now

If we pass in a hard bonud function using the `.bind` method: `setTimeout(workshop.ask.bind(workshop),10,"hard bound this?")`... it will take away that whole flexibility thing and force it to **only** use the `this` that we've specified on this line

`setTimeout(workshop.ask.bind(workshop),10,"hard bound this?")` -> we're saying invoke this function, and no matter how you invoke it, always use `workshop` as its `this` context

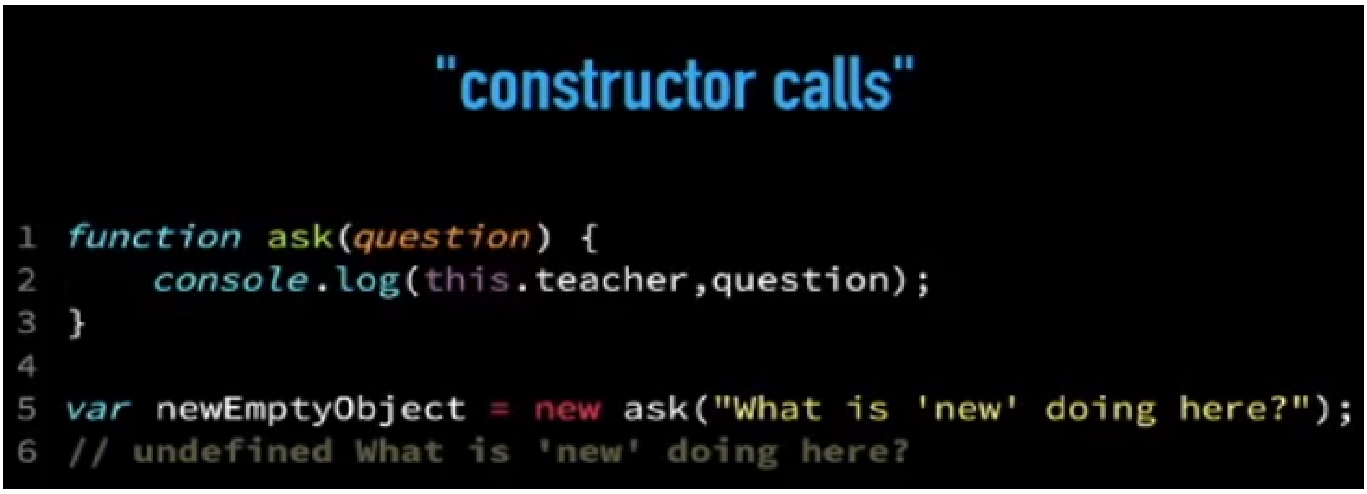
In other words the `.bind` method, it doesn't invoke the function, it produces a new function which is bound to a particular specific `this` context

So there's a tradeoff... we have a predictabe, flexible `this` binding, but then you see some scenarios where it's kind of frustrating that it's flexible **AND** what I'd really like is for it to be super predictable

`new` keyword

The **3rd** way we'll look at invoking functions uses the `new` keyword And I fully understand that the `new` keyword seems as if it has something to do with invoking class constructors **It has nothing to do with invoking class constructors**, It's just an unfortunate syntactic trick to make it look like it's dealing with classes when it's really not **Actually,** the `new` keyword is the third way that you can invoke a function, and it does four very specific things, which aren't very obvious

But the **purpose** of the `new` keyword is actually to invoke a function with a `this` keyword pointing at a whole new empty object



If we have invoking functions and pointing them at a context object like a `workshop.ask` (thats 1 way) **or** we say I'm gonna invoke a function and give it a specific object with a `.call()` or `.apply()` or I'm gonna force it with a `.bind()` (thats a 2nd way)

A **third way** of doing it is to say I wanna invoke a function and use a whole new **empty object** And the `new` keyword can accomplish that..

The `new` keyword does other stuff, but it also accomplishes that task (which is to invoke a function with a new empty object) Now I would that you could accomplish that same goal by saying: `arbitraryFunc.call{/../}` cause that would our function in the context of a brand new empty object So the `new` keyword isn't actually buying you much except the syntactic sugar of... "Hey I want this function invoked with a new `this` context

BUT what are the four things that the `new` keyword is going to do when it's used to invoke our function?

1. We create a brand new empty object
2. the `new` keyword links that object to another object
3. It invokes the function, with its `this` keyword pointed at the new object (not the linked object)
4. The `new` keyword after the function call is done, if that function does not return its own object, the `new` keyword assumes that you meant to return `this` keyword

These 4 things happen every single time a `new` keyword is invoked with a function

Even if you put an object after `new` it will still do these 4 things

Final way of invoking a function: **Default Binding**

```
var teacher = "Kyle";

fuction ask(question) {
  console.log(this.teacher,question)
}

function askAgain(question) {
  "use strict";
```

```
  console.log(this.teacher, question);
}

ask("what's the non-strict mode default?")
// Kyle What's the non-strict-mode default?

ask("What's the strict-mode default?")
// TypeError
```

`ask("what's the non-strict mode default?")` - notice how in this line it **does not** follow any of the other rules:

- I don't have any context object
- I don't have any `.call()` or `.bind()`
- I don't have a `new` keyword

It's just a plain old function call, **it doesn't match any of the other rules** So since it doesn't match any of the other rules, the fallback is defined in the spec as, in non-strict mode, **default to the global** That's why we print "Kyle" because there's a global variable called `teacher`

Binding Precedence

```
var workshop = {
  teacher: "Kyle",
  ask: function ask(question) {
    console.log(this.teacher,question)
  },
};

new (workshop.ask.bind(workshop))("what does this do?");
// undefined what does this do?
```

What if we have crazy code like this, what binding rule (of the 4) takes precedence?

So from here forward, if you ever need to ask yourself **what is my `this` keyword going to point at when this function get invoked**...this is how you determine it:

1. Is the function called by `new`?
2. Is the function called by `call()` or `apply()`?
   - Note `bind()` effectively uses `apply()`
3. Is the function called on a context object?
4. **Default**: global object