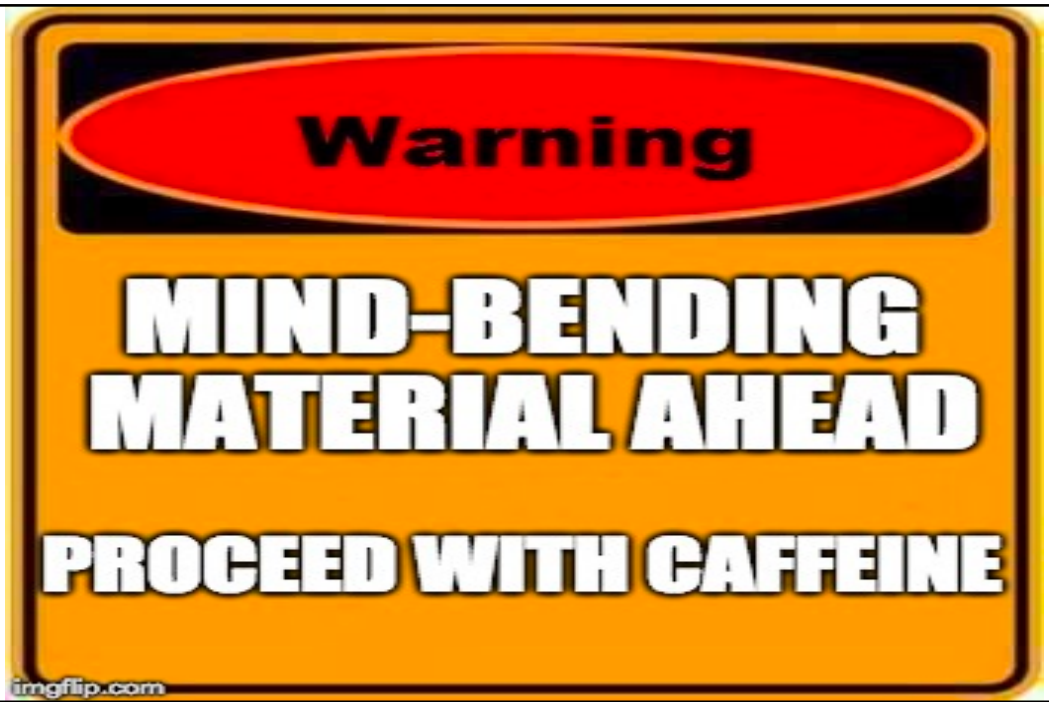
Understanding execution context

What is *this*?



tl;dr

- Context may change the way a non-pure function executes
- And how a function is executed causes that function to look at different contexts
- There are three ways to execute a function: Method form, function form, and explicit binding form
- Function.prototype.bind() allows you to bind an external context to a function
- Even how you define the function will change its context when it runs. Arrow functions use their parents' context.

The *this* variable refers to the current context (aka. environment)

Costello: My code doesn't work
Abbott: I see the problem. *this* is wrong.
Costello: I know. What's the issue?
Abbott: *this*. You're misusing *this*
Costello: I'm misusing what?
Abbott: No, you're misusing *this*.
Costello: Well are you going to tell me what's the problem or aren't you?
Abbott: I am. You're misusing *this*.
Costello: Exactly where's the issue in my code?
Abbott: No, not where ... *this*.
...

Meaning depends on context. For instance ...

Bad	Good
• His feelings were *crushed* when she left. • Is it me or was that waiter *salty* to us?	• He *crushed* it in the 4th quarter! • The bisque was delicious! So *salty*!

How you invoke a function changes *this*

3 ways to invoke a function

1. Method form
2. Call/apply form
3. Function form

Say you have a getName() function

```
function getName(format="casual") {
  let n = "";
  if (format==="formal")
    n=`${this.last}, ${this.first} ${this.middle}`;
  else
    n=`${this.first} ${this.first}`;
  return n;
}
```

- How well it runs depends on what *this* is

The Function Form

... just ... uh ... call it

When you just call it, this refers to the global object

- Example:

```
getName();
```

- Fails b/c window isn't a cart. "What's a last?" "What's a first?" "What's a middle?"

Rule of thumb ... if *this* appears, it was never intended to be called as a simple function

We can set/affix *this* with the bind() method

- When you pass in a context to bind, it creates a new function with a bound context.

```
const p = {
  first:"George",
  middle:"Oscar",
  last:"Bluth" };
const gn = getName.bind(p);
const name = gn();
console.log(name);
```

- // George Bluth

Call/apply form

Explicit binding

With call and apply, the context is passed in to the function

We're explicitly telling the function what it's *this* will be

For example...

```
let p1 = {first:"Lucille", last:"Austero"};
let name1 = getName.call(p1, "formal");
let name1 = getName.apply(p1, ["casual"]);
```

Syntax:

- function.call(context, parm1, parm2 ...)
  - Pass in any number of parameters
- function.apply(context, [parm1, parm2 ...])
  - Pass in an array of parameters

Summary

Form	Example	this
Method	obj1.foo();	obj1
Function	foo()	global
Call/apply	foo.call(obj1, p1, p2, ...); foo.apply(obj1, [p1, p2, ...]);	obj1

Setting context during definition

Let's illustrate the point ...

```
this.name = "Global object";
const f = function () {
  console.log(`I am ${this.name}`);
}
const lawyer = {
  name:"Bob Boblaw", getName:f
}
lawyer.getName();
```

- What would you expect to be console.logged?

If it binds to the **definer** it'll say ...

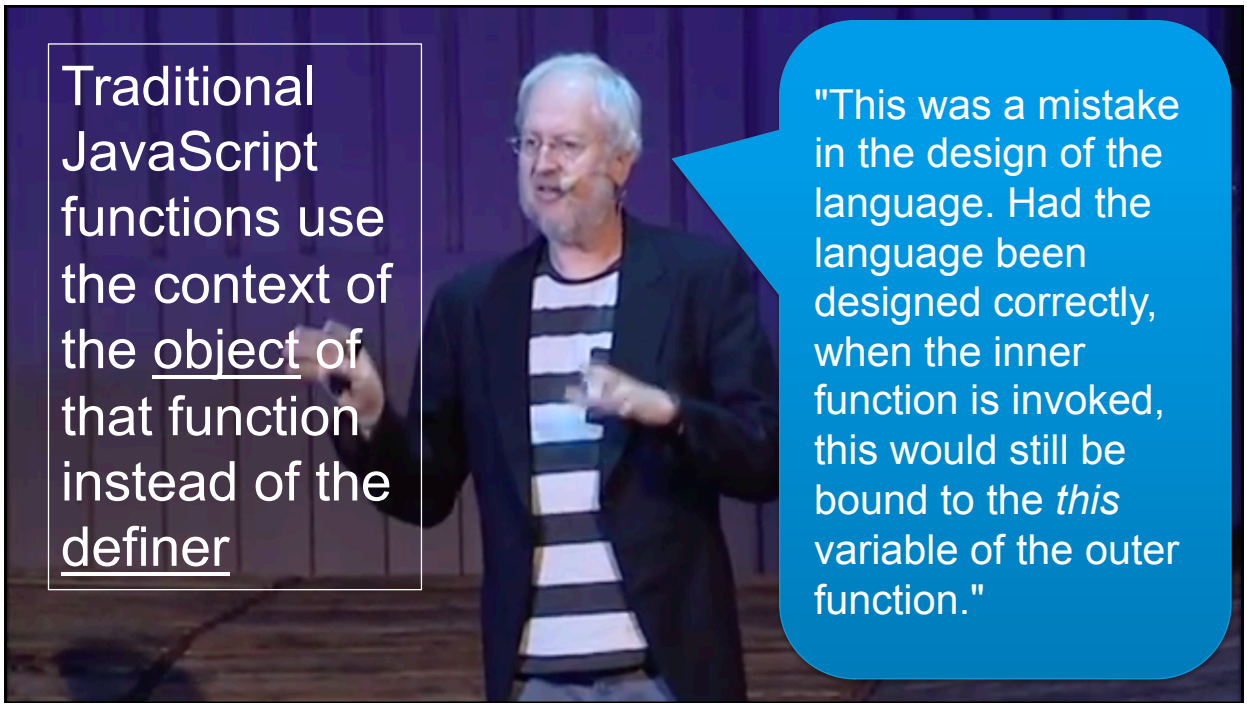
_____

(aka "lexical scope")

If it binds to its current **object** it'll say ...

_____

Do you see how this can be surprising?



This was 'fixed' in ES2015 with arrow functions.

Arrow functions bind lexically

```
this.name = "Global object";
const getName = () => {
  console.log(`I am ${this.name}`);
}
const lawyer = {
  name:"Bob Boblaw", getName:getName
}
lawyer.getName();
```

tl;dr

- Context may change the way a non-pure function executes
- And how a function is executed causes that function to look at different contexts
- There are three ways to execute a function: Method form, function form, and explicit binding form
- Function.prototype.bind() allows you to bind an external context to a function
- Even how you define the function will change its context when it runs. Arrow functions use their parents' context.