ADVANCED JS

```javascript
catName("Tom");

function catName(name) {
  console.log("My cat's name is " + name);
}
/*
The result of the code above is: "My cat's name is
Tom"
*/
```

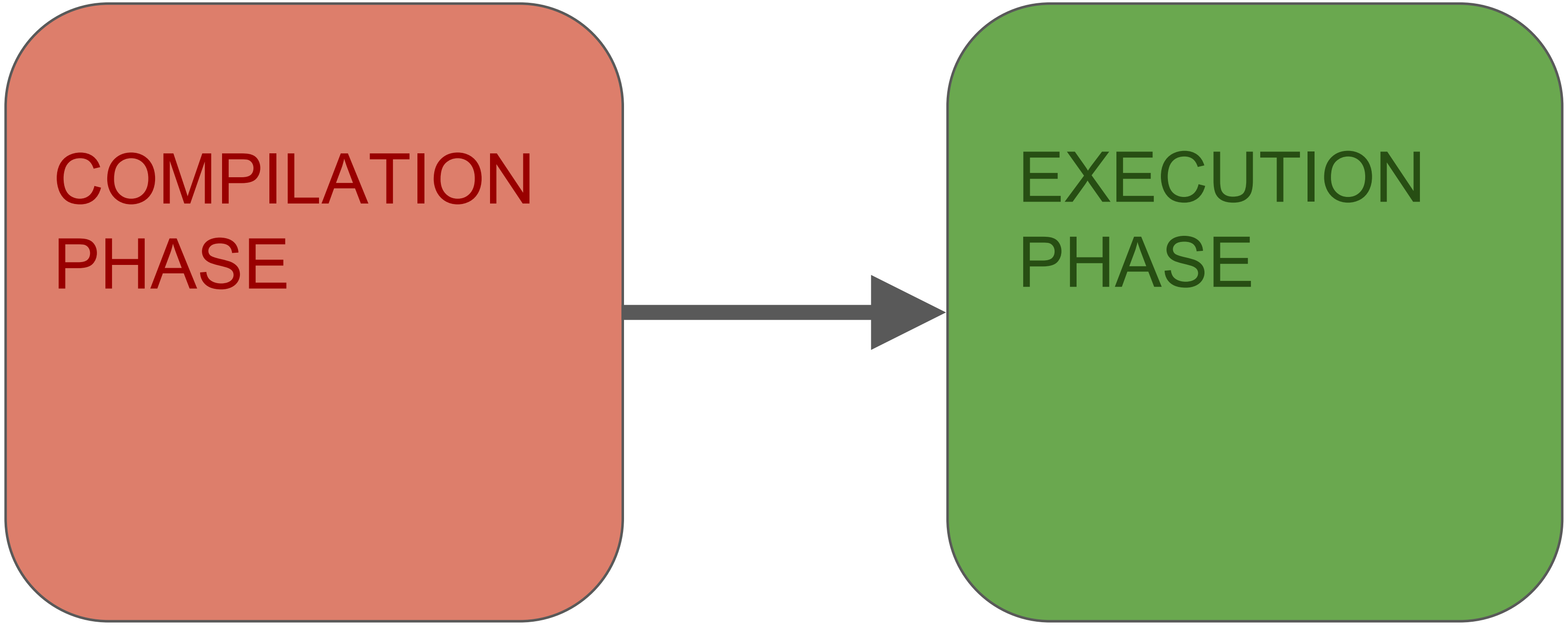
```
function(){

  var a = true;
  var b = true;

  var c = function(){
    return true;
  };

  function d(){
    return true;
  };

}();
```

```
function(){

  function d(){
    return true;
  };
  var a;
  var b;
  var c;

  a = true;
  b = true;

  c = function(){
    return true;
  };

}();
```

Oh!! Hoisting is very simple. You know it is same as Flag hoisting :P

Is JS a compiled language/ interpreted?

Its JIT(Just In Time)



```
var foo = "bar";
var foo = "bar";

function simple() {
  var foo = "baz";
  console.log(foo);
}

simple();
console.log(foo);
```

Let’s play a game for a while!

```
a;
b;
var a=2;
var b=a;
b;
a;
```

Let’s apply what we have learnt!

```
expression(); //Output: "TypeError: expression is not a
function

var expression = function() {
  console.log('Will this work?');
};
```

I can... but I won't

```
var a = b();
var c = d();
a;
c;

function b(){
return c;
}

var d  =
function(){
            return b();
}
```

Mutual recursion!

```
a(1);

function a(foo){
        if(foo>20) return foo;
        return b(foo+2);
}

function b(foo){
        return c(foo)+1;
}

function c(foo){
        return a(foo*2);
}
```

Quick Trivia!

```
var foo = "bar";

function bar() {
        var foo = "baz";
        function baz(foo) {
                foo = "bam";
                bam = "yay";
        }
        baz();
}
bar();
foo; // "bar”
bam; // "yay”
baz(); // ReferenceError
```

Global leak!

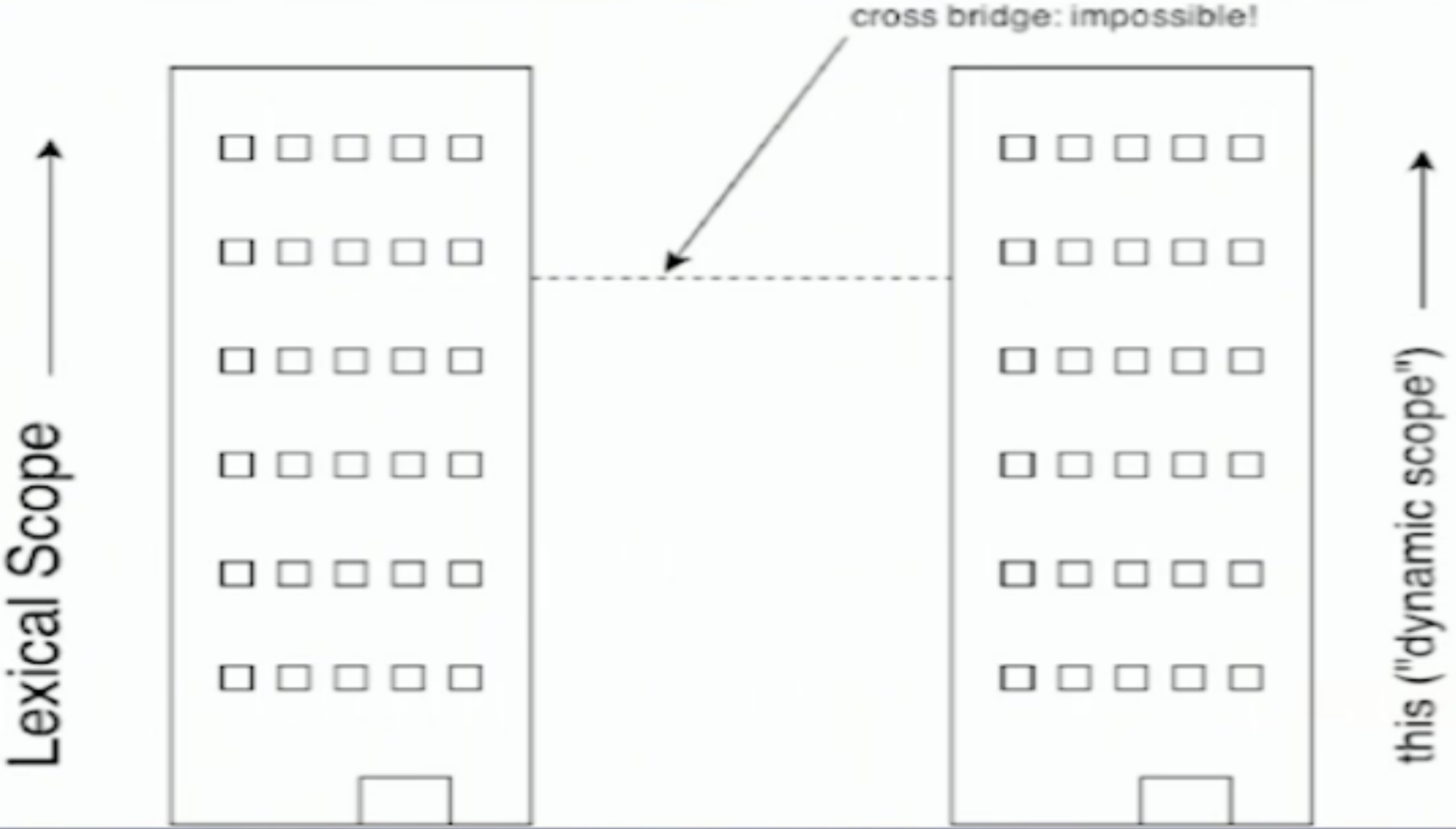
Lets see what MDN has to say about this!

https://developer.mozilla.org/en-US/docs/Glossary/Hoisting

Javascript has function scope only*

```
var foo;
try{
foo.length;
}
catch(err){
console.log(err); //TypeError
}
console.log(err); //ReferenceError
```

LEXICAL SCOPE
VS
DYNAMIC SCOPE


cross bridge: impossible!
Lexical Scope
this ("dynamic scope")

```
// theoretical dynamic scoping
function foo(){
	console.log(bar); //dynamic!
}
function baz(){
	var bar = "bar";
	foo();
}
baz();
```

Dynamic scope!

Cheating lexical scope!

```
var bar = "bar";

function foo(str){
eval(str); //cheating
console.log(bar); //42
}

foo("var bar=42;");
```

eval keyword!

```
var bar = "bar";

function foo(str){
eval(str); //cheating
console.log(bar); //42
}

foo("var bar=42;");
```

Even worse way to cheat!

IIFE PATTERN

```
var foo = "foo";

(function(){

var foo = "foo2";

console.log(foo); // "foo2”

})();

console.log(foo); //"foo"
```

let (ES6+)

```
function foo(bar){
        if(bar){
        console.log(baz);
        //ReferenceError
        let baz = bar;  }
}
foo("bar");
```

Temporal dead zone!

“this” keyword

It all depends on the call site

4th rule(Default binding rule)

```
 function foo(bar){

                console.log(this.bar);

}

var bar = "bar1";

var o2 = {bar: "bar2", foo: foo};

var o3 = {bar: "bar3", foo: foo};

foo(); //"bar1”

o2.foo(); //"bar2”

o3.foo(); //"bar3"
```

3rd rule(Implicit binding rule)

```
 function foo(bar){

          console.log(this.bar);

}

var bar = "bar1";

var o2 = {bar: "bar2", foo: foo};

var o3 = {bar: "bar3", foo: foo};

foo(); //"bar1”
```

o2.foo(); //"bar2”

```
o3.foo(); //"bar3"
```

```
var o1 = {

bar: "bar1",

foo: function(){console.log(this.bar);}

};

var o2 = {bar: "bar2", foo: o1.foo};

var bar = "bar3";

var foo = o1.foo;

o1.foo(); //bar1

o2.foo(); //bar2

foo(); //bar3
```

2nd rule(Explicit binding rule)

```
function foo(){
        console.log(this.bar);
}
var bar = "bar1";
var obj = {bar: "bar2"};


foo();  //bar1
foo.call(obj); //bar2
```

Hard Binding!

```
function foo(){

console.log(this.bar);

}

var obj = {bar:"bar"};

var obj2 = {bar: "bar2"};

var orig = foo;

foo = function(){ orig.call(obj); };

foo(); //bar

foo.call(obj2); //bar
```

Bind utility

```
function bind(fn,o){

return function(){ fn.call(o); };

}

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

var obj = {bar:"bar"};

var obj2 = {bar: "bar2"};

foo = bind(foo,obj);

foo(); //bar

foo.call(obj2); //bar
```

1st rule(New keyword)

```
function foo(){
        this.bar = "baz";
        console.log(this.bar + " " + baz);
}
var bar = "bar";
var baz = new foo();
```

3 things happen

- A brand new empty object will be created out of thin air.
- The brand new spoof object gets bound as the “this” keyword for the purposes of that function call.
- If that function otherwise does not return anything then it will implicitly insert a “return this”, so that brand new spoof object will be implicitly returned for us.

4 rules

1. Was the function called with the new keyword?
2. Was the function called with ‘call’ or ‘apply’ specifying an explicit this?
3. Was a function called via a containing/owning object(context)
4. DEFAULT: global object(except strict mode)

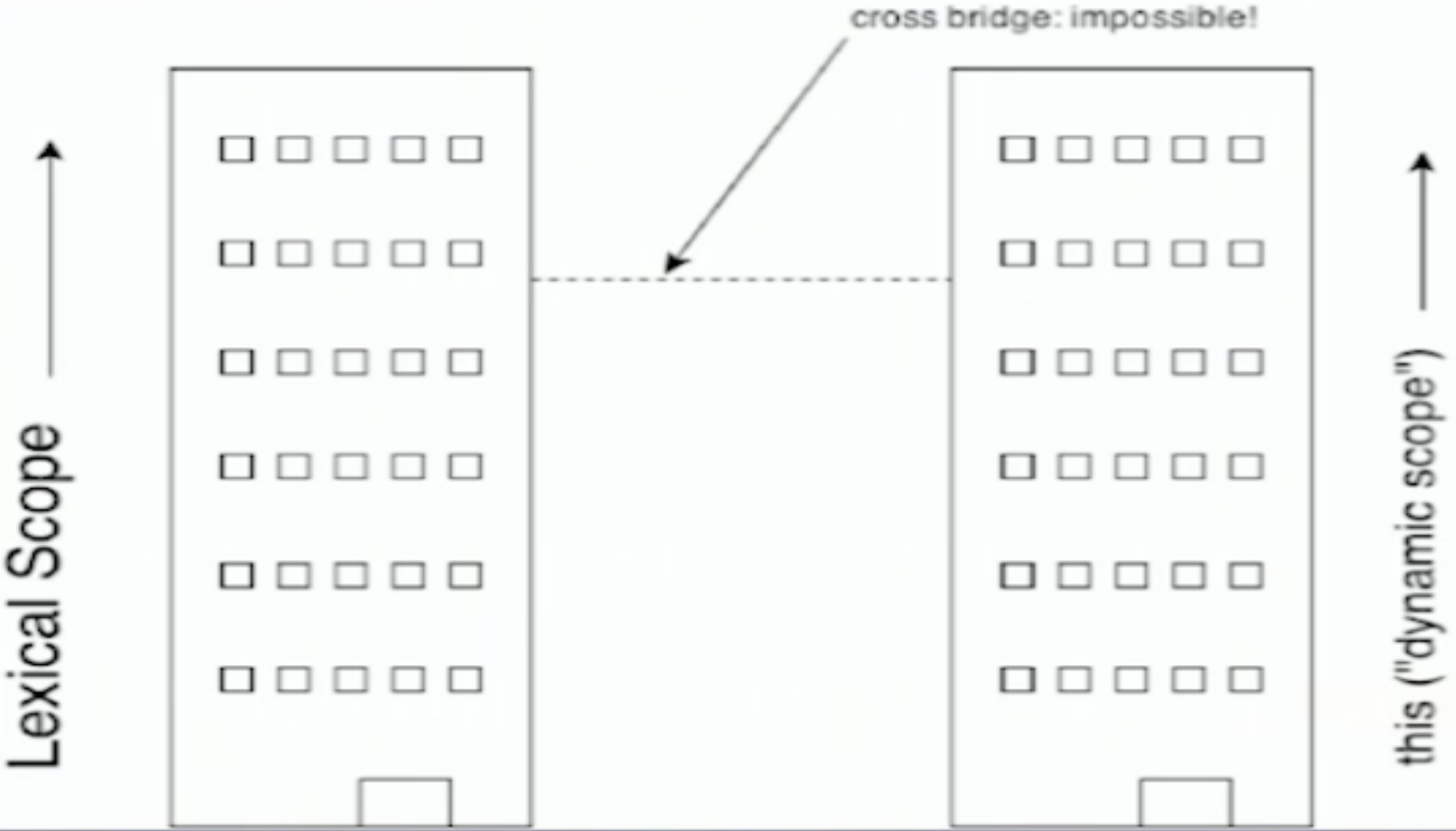
Binding confusion!

Problem statement

```
function foo(){

var bar = "bar1";

baz();}

function baz(){

console.log(this.bar);

}

var bar = "bar2";

foo();
```

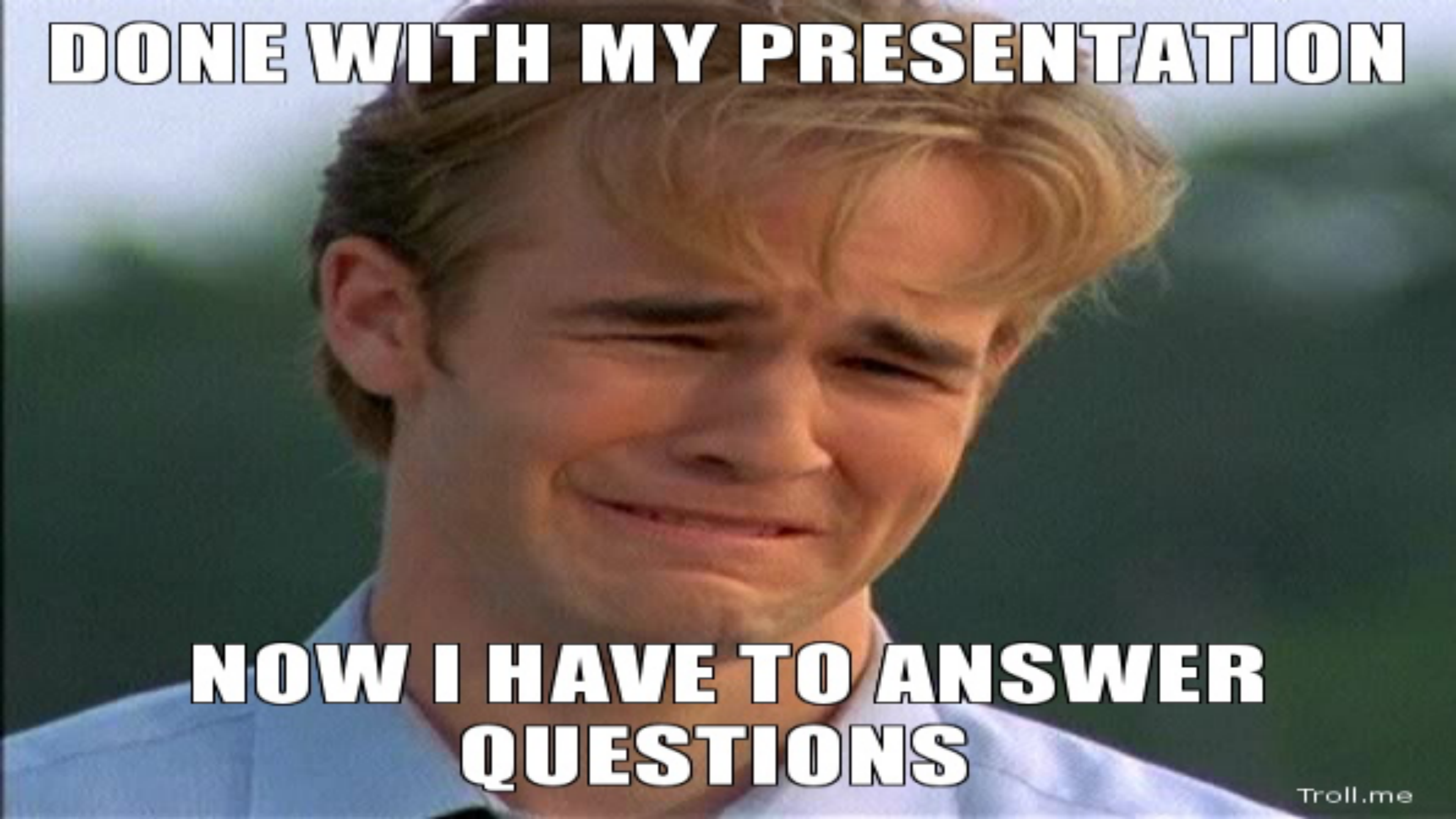
Incorrect solution

```
function foo(){

var bar = "bar1";

this.baz = baz;

this.baz();}

function baz(){

console.log(this.bar);}

var bar = "bar2";

foo(); //refers to global bar and not local bar in foo
```


cross bridge: impossible!
Lexical Scope
this ("dynamic scope")

ECMAScript SPEC

http://www.ecma-international.org/ecma-262/5.1/


DONE WITH MY PRESENTATION
NOW I HAVE TO ANSWER QUESTIONS
Troll.me