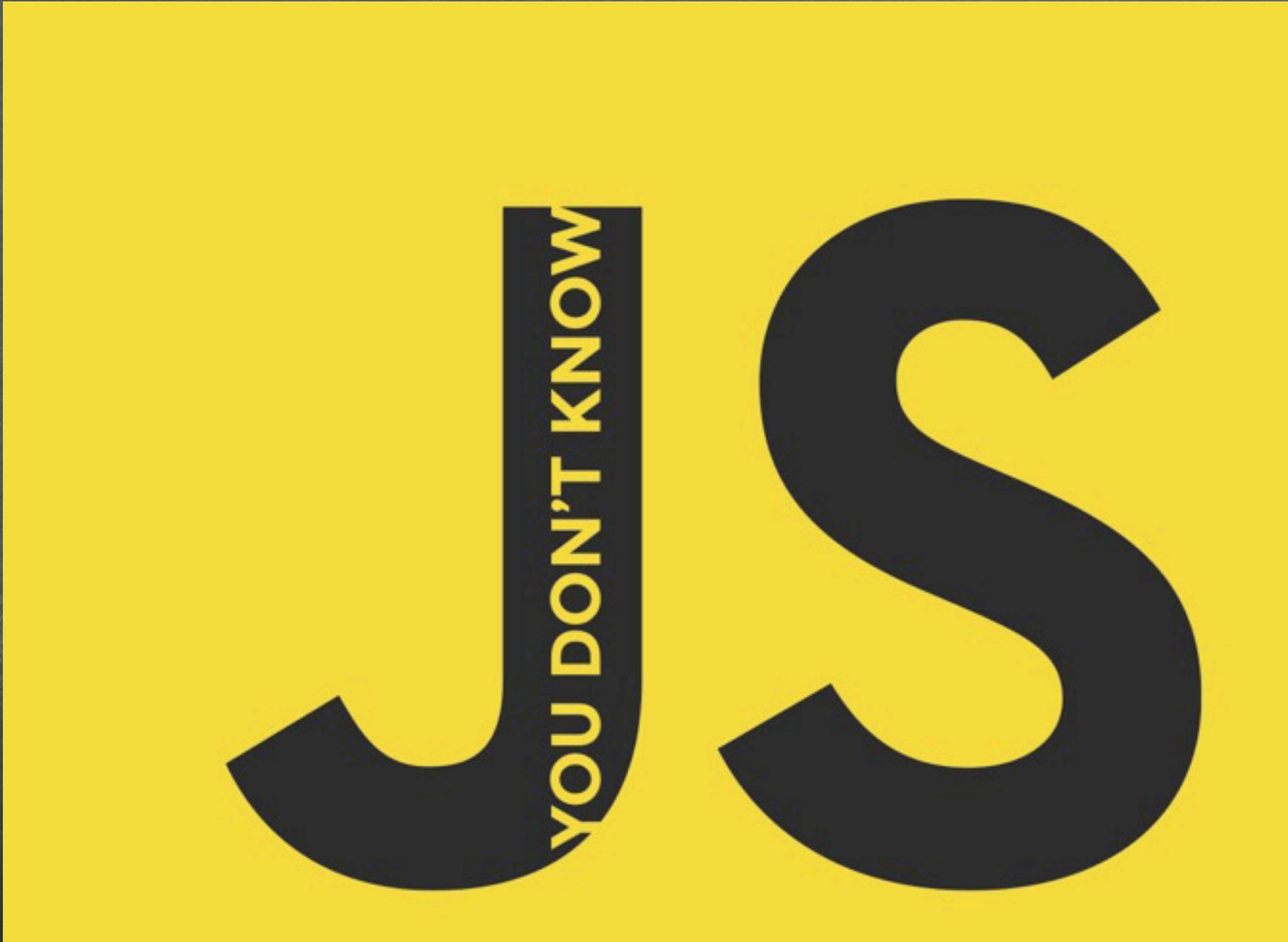


Kyle Simpson
@getify
<http://getify.me>

- LABjs
- grips
- asynquence





<http://YouDontKnowJS.com>

Agenda

Scope, Closures

- Nested Scope
- Hoisting
- Closure

Scope: where to look
for things

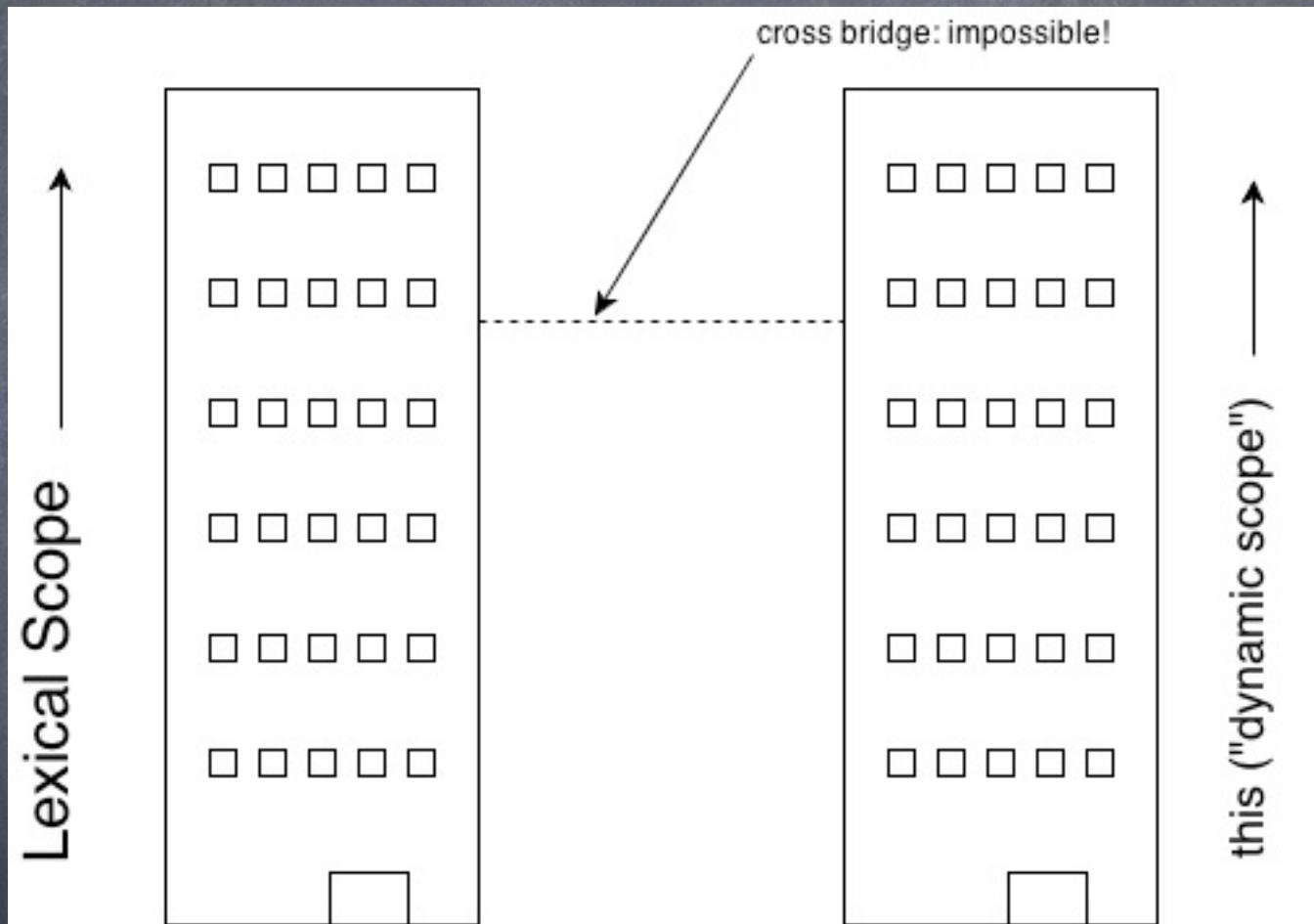
Scope

JavaScript has
function scope only*

```
1 var foo = "bar";  
2  
3 function bar() {  
4     var foo = "baz";  
5 }  
6  
7 function baz(foo) {  
8     foo = "bam";  
9     bam = "yay";  
10 }
```

```
1 var foo = "bar";
2
3 function bar() {
4     var foo = "baz";
5
6     function baz(foo) {
7         foo = "bam";
8         bam = "yay";
9     }
10    baz();
11 }
12
13 bar();
14 foo;           // ???
15 bam;          // ???
16 baz();        // ???
```

```
1 var foo = "bar";
2
3 function bar() {
4     var foo = "baz";
5
6     function baz(foo) {
7         foo = "bam";
8         bam = "yay";
9     }
10    baz();
11 }
12
13 bar();
14 foo;           // "bar"
15 bam;          // "yay"
16 baz();        // Error!
```



Scope

```
1 var foo = function bar() {  
2     var foo = "baz";  
3  
4     function baz(foo) {  
5         foo = bar;  
6         foo; // function...  
7     }  
8     baz();  
9 };  
10  
11 foo();  
12 bar(); // Error!
```

Scope: function scope?

```
1 var foo;  
2  
3 try {  
4     foo.length;  
5 }  
6 catch (err) {  
7     console.log(err); // TypeError  
8 }  
9  
10 console.log(err); // ReferenceError
```

Scope: block scope?

lexical scope

dynamic scope

Scope

```
1 function foo() {  
2   var bar = "bar";  
3  
4   function baz() {  
5     console.log(bar); // lexical!  
6   }  
7   baz();  
8 }  
9 foo();
```

Scope: lexical

```
1 var bar = "bar";
2
3 function foo(str) {
4     eval(str); // cheating!
5     console.log(bar); // 42
6 }
7
8 foo("var bar = 42;");
```

Scope: WAT!?

```
1 var obj = {  
2     a: 2,  
3     b: 3,  
4     c: 4  
5 };  
6  
7 obj.a = obj.b + obj.c;  
8 obj.c = obj.b - obj.a;  
9  
10 with (obj) {  
11     a = b + c;  
12     c = b - a;  
13     d = 3; // <-- Look!  
14 }  
15  
16 obj.d; // undefined  
17 d; // 3 -- oops!
```

IIFE

```
1 var foo = "foo";
2
3 (function(){
4
5     var foo = "foo2";
6     console.log(foo);    // "foo2"
7
8 })();
9
10 console.log(foo);   // "foo"
```

<http://benalman.com/news/2010/11/immediately-invoked-function-expression/>

Function Scope

```
1 var foo = "foo";
2
3 (function(bar){
4
5     var foo = bar;
6     console.log(foo);    // "foo"
7
8 })(foo);
9
10 console.log(foo);   // "foo"
```

Block Scope

let (ES6+)

Block Scope

```
1 function foo() {  
2     var bar = "bar";  
3     for (let i=0; i<bar.length; i++) {  
4         console.log(bar.charAt(i));  
5     }  
6     console.log(i); // ReferenceError  
7 }  
8  
9 foo();
```

Block Scope: let

```
1 function foo(bar) {  
2     if (bar) {  
3         let baz = bar;  
4         if (baz) {  
5             let bam = baz;  
6         }  
7         console.log(bam); // Error  
8     }  
9     console.log(baz); // Error  
10 }  
11  
12 foo("bar");
```

Block Scope: let

```
1 function foo(bar) {  
2     let (baz = bar) {  
3         console.log(baz); // "bar"  
4     }  
5     console.log(baz); // Error  
6 }  
7  
8 foo("bar");
```

Block Scope: let

```
1 function foo(bar) {  
2     /*let*/ { let baz = bar;  
3         console.log(baz); // "bar"  
4     }  
5     console.log(baz); // Error  
6 }  
7  
8 foo("bar");
```

<https://gist.github.com/getify/5285514>

Block Scope: let

dynamic scope

Scope

```
1 // theoretical dynamic scoping
2 function foo() {
3     console.log(bar); // dynamic!
4 }
5
6 function baz() {
7     var bar = "bar";
8     foo();
9 }
10
11 baz();
```

Scope: dynamic!?!?

Quiz

1. What type of scoping rule(s) does JavaScript have? Exceptions?
2. What are the different ways you can create a new scope?
3. What's the difference between undeclared and undefined?

Scope: hoisting

```
1 a;           // ???
2 b;           // ???
3 var a = b;
4 var b = 2;
5 b;           // 2
6 a;           // ???
```

Scope: hoisting

```
1 var a;  
2 var b;  
3 a;           // ???  
4 b;           // ???  
5 a = b;  
6 b = 2;  
7 b;           // 2  
8 a;           // ???
```

Scope: hoisting

```
1 var a = b();  
2 var c = d();  
3 a;           // ???  
4 c;           // ???  
5  
6 function b() {  
7     return c;  
8 }  
9  
10 var d = function() {  
11     return b();  
12 };
```

Scope: hoisting

```
1 function b() {  
2     return c;  
3 }  
4 var a;  
5 var c;  
6 var d;  
7 a = b();  
8 c = d();  
9 a;           // ???  
10 c;          // ???  
11 d = function() {  
12     return b();  
13 };
```

Scope: hoisting

```
1 a(1);          // ???
2
3 function a(foo) {
4     if (foo > 20) return foo;
5     return b(foo+2);
6 }
7 function b(foo) {
8     return c(foo) + 1;
9 }
10 function c(foo) {
11     return a(foo*2);
12 }
```

Hoisting: recursion

```
1 function foo(bar) {  
2     if (bar) {  
3         console.log(baz); // ReferenceError  
4         let baz = bar;  
5     }  
6 }  
7  
8 foo("bar");
```

Hoisting: **let** gotcha

Closure

Closure is when a function
“remembers” its lexical scope
even when the function is
executed outside that lexical
scope.

```
1 function foo() {  
2     var bar = "bar";  
3  
4     function baz() {  
5         console.log(bar);  
6     }  
7  
8     bam(baz);  
9 }  
10  
11 function bam(baz) {  
12     baz(); // "bar"  
13 }  
14  
15 foo();
```

```
1 function foo() {  
2     var bar = "bar";  
3  
4     return function() {  
5         console.log(bar);  
6     };  
7 }  
8  
9 function bam() {  
10    foo()(); // "bar"  
11 }  
12  
13 bam();
```

```
1 function foo() {  
2     var bar = "bar";  
3  
4     setTimeout(function() {  
5         console.log(bar);  
6     }, 1000);  
7 }  
8  
9 foo();
```

```
1 function foo() {  
2     var bar = "bar";  
3  
4     $("#btn").click(function(evt) {  
5         console.log(bar);  
6     });  
7 }  
8  
9 foo();
```

```
1 function foo() {  
2     var bar = 0;  
3  
4     setTimeout(function(){  
5         console.log(bar++);  
6     },100);  
7     setTimeout(function(){  
8         console.log(bar++);  
9     },200);  
10 }  
11  
12 foo(); // 0 1
```

Closure: shared scope

```
1 function foo() {  
2     var bar = 0;  
3  
4     setTimeout(function(){  
5         var baz = 1;  
6         console.log(bar++);  
7  
8         setTimeout(function(){  
9             console.log(bar+baz);  
10        },200);  
11    },100);  
12 }  
13  
14 foo(); // 0 2
```

Closure: nested scope

```
1 for (var i=1; i<=5; i++) {  
2     setTimeout(function(){  
3         console.log("i: " + i);  
4     }, i*1000);  
5 }
```

Closure: loops

```
1 for (var i=1; i<=5; i++) {  
2   (function(i){  
3     setTimeout(function(){  
4       console.log("i: " + i);  
5     }, i*1000);  
6   })(i);  
7 }
```

Closure: loops

```
1 for (let i=1; i<5; i++) {  
2     setTimeout(function(){  
3         console.log("i: " + i);  
4     }, i*1000);  
5 }
```

Closure: loops + block scope

```
1 var foo = (function(){
2
3     var o = { bar: "bar" };
4
5     return { obj: o };
6
7 })();
8
9 console.log(foo.obj.bar);    // "bar"
```

Closure?

```
1 var foo = (function(){
2
3     var o = { bar: "bar" };
4
5     return {
6         bar: function(){
7             console.log(o.bar);
8         }
9     };
10
11 })();
12
13 foo.bar();           // "bar"
```

Closure: classic module pattern

```
1 var foo = (function(){
2     var publicAPI = {
3         bar: function(){
4             publicAPI.baz();
5         },
6         baz: function(){
7             console.log("baz");
8         }
9     };
10    return publicAPI;
11 })();
12
13 foo.bar(); // "baz"
```

Closure: modified module pattern

```
1 define("foo", function(){
2
3     var o = { bar: "bar" };
4
5     return {
6         bar: function(){
7             console.log(o.bar);
8         }
9     };
10
11});
```

Closure: modern module pattern

foo.js:

```
1 var o = { bar: "bar" };
2
3 export function bar() {
4     return o.bar;
5 }
```

```
1 import { bar } from "foo";
2
3 bar();          // "bar"
4
5 import * as foo from "foo";
6
7 foo.bar();    // "bar"
```

Closure: ES6+ module pattern

Quiz

1. What is a closure and how is it created?
2. How long does its scope stay around?
3. Why doesn't a function callback inside a loop behave as expected? How do we fix it?
4. How do you use a closure to create an encapsulated module? What's the benefits of that approach?

(exercise #1: 20min)

<http://dmitrysoshnikov.com/ecmascript/javascript-the-core/>

Agenda

Object-Orienting

- this
- Encapsulation
- Common OO Patterns
- prototype
- “Inheritance” vs. “Behavior Delegation”
(OO vs. OLOO)

this

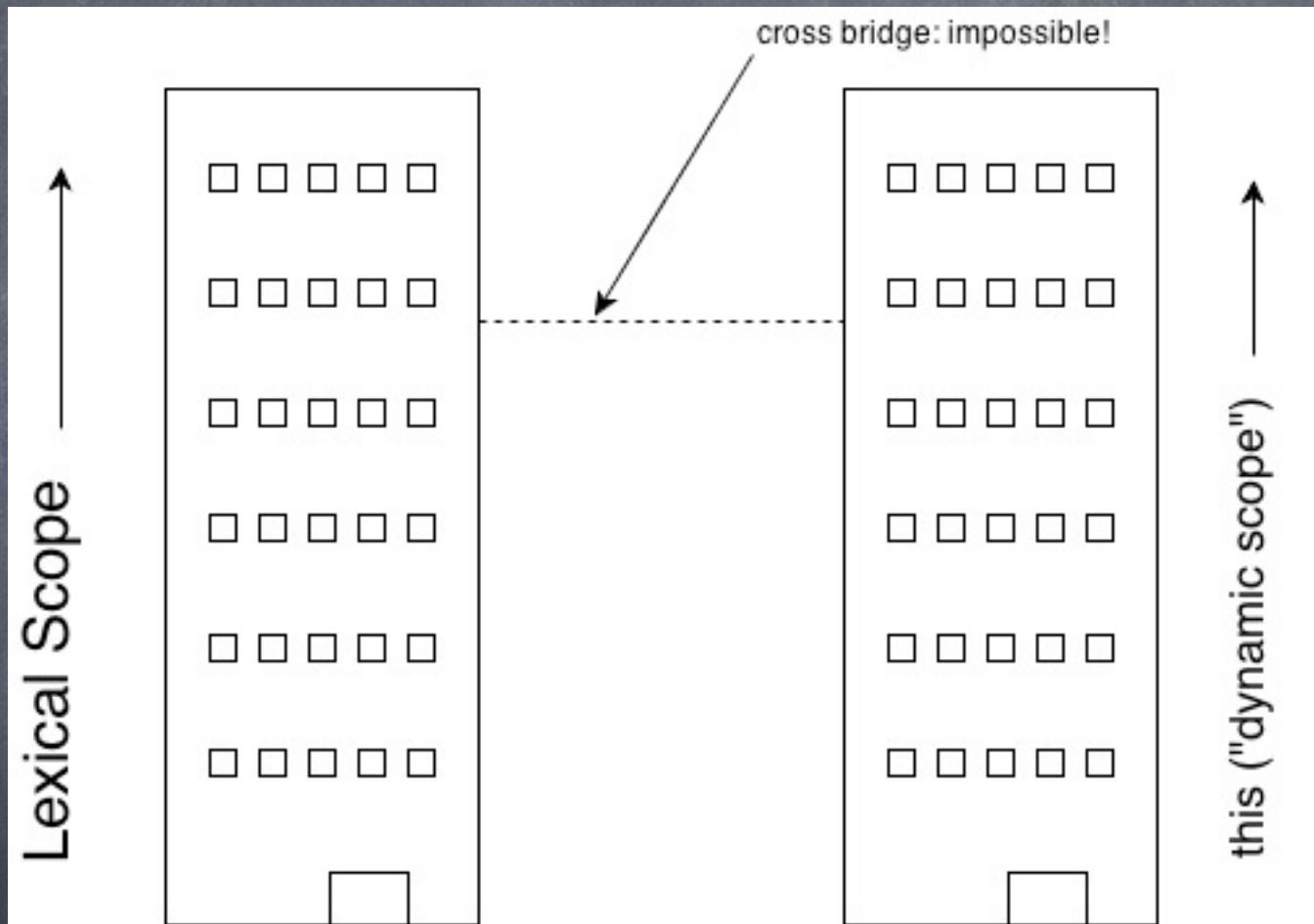
Every function, while executing, has a reference to its current execution context, called **this**.

this

Remember lexical scope vs.
dynamic scope?

JavaScript's version of
“dynamic scope” is **this**.

this



Scope

```
1 function foo() {  
2     console.log(this.bar);  
3 }  
4  
5 var bar = "bar1";  
6 var o2 = { bar: "bar2", foo: foo };  
7 var o3 = { bar: "bar3", foo: foo };  
8  
9 foo(); // "bar1"  
10 o2.foo(); // "bar2"  
11 o3.foo(); // "bar3"
```

this: implicit & default binding

```
1 var o1 = {  
2     bar: "bar1",  
3     foo: function() {  
4         console.log(this.bar);  
5     }  
6 };  
7 var o2 = { bar: "bar2", foo: o1.foo };  
8  
9 var bar = "bar3";  
10 var foo = o1.foo;  
11  
12 o1.foo();           // "bar1"  
13 o2.foo();           // "bar2"  
14 foo();              // "bar3"
```

this: implicit & default binding

```
1 function foo() {  
2     console.log(this.bar);  
3 }  
4  
5 var bar = "bar1";  
6 var obj = { bar: "bar2" };  
7  
8 foo();           // "bar1"  
9 foo.call(obj);  // "bar2"
```

this: explicit binding

```
1 function foo() {  
2     console.log(this.bar);  
3 }  
4  
5 var obj = { bar: "bar" };  
6 var obj2 = { bar: "bar2" };  
7  
8 var orig = foo;  
9 foo = function(){ orig.call(obj); };  
10  
11 foo(); // "bar"  
12 foo.call(obj2); // ???
```

this: hard binding

```
1 function bind(fn,o) {  
2     return function() {  
3         fn.call(o);  
4     };  
5 }  
6 function foo() {  
7     console.log(this.bar);  
8 }  
9  
10 var obj = { bar: "bar" };  
11 var obj2 = { bar: "bar2" };  
12  
13 foo = bind(foo,obj);  
14  
15 foo();           // "bar"  
16 foo.call(obj2); // ???
```

this: hard binding

https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Function/bind

```
1 function foo(baz,bam) {  
2     console.log(this.bar + " " + baz +  
3                 " " + bam);  
4 }  
5  
6 var obj = { bar: "bar" };  
7 foo = foo.bind(obj,"baz"); // ES5 only!  
8  
9 foo("bam");           // "bar baz bam"
```

this: hard binding

```
1 function foo() {  
2     this.baz = "baz";  
3     console.log(this.bar + " " + baz);  
4 }  
5  
6 var bar = "bar";  
7 var baz = new foo(); // ???
```

this: new

```
1 function something() {  
2     this.hello = "hello";  
3     console.log(this.hello, this.who);  
4 }  
5  
6 var who = "global", foobar, bazbam,  
7     obj1 = { who: "obj1", something: something },  
8     obj2 = { who: "obj2" };  
9  
10 something(); // "hello" "global"  
11 console.log(hello); // "hello"           <-- OOPS!!!  
12  
13 obj1.something(); // "hello" "obj1"  
14 console.log(obj1.hello); // "hello"  
15  
16 obj1.something.call(obj2); // "hello" "obj2"  
17 console.log(obj2.hello); // "hello"  
18  
19 foobar = something.bind(obj2);  
20 foobar(); // "hello" "obj2"  
21 foobar.call(obj1); // "hello" "obj2" <-- STILL "obj2"  
22  
23 bazbam = new something(); // "hello" undefined  
24 console.log(bazbam.hello); // "hello"  
25  
26 bazbam = new obj1.something(); // "hello" undefined  
27 bazbam = new foobar(); // "hello" undefined <-- LOOK, not "obj2"
```

this: order of precedence

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

this: determination

Quiz

1. What determines which object a function's **this** points to? What's the default?
2. How do you "borrow" a function by implicit assignment of **this**?
3. How do you explicitly bind **this**?
4. How can you force a specific **this** for a function? Why do that? Why not?
5. How do you create a new **this**?

this

Information hiding

Encapsulation

Principle of Least Exposure

Encapsulation

In OO, we “hide” private
data from the external API

Encapsulation

private
protected
public

Encapsulation

```
1 var Hello = function(d){  
2     var name; // private data!  
3  
4     // auto-execute "constructor"  
5     (function Hello(d) {  
6         name = d.name;  
7     })(d);  
8  
9     function speak() {  
10        console.log("Hello " + name);  
11    }  
12  
13     var publicAPI = {  
14         speak: speak  
15     };  
16     return publicAPI;  
17 };  
18  
19 var world = Hello({ name:"world" });  
20 world.speak(); // "Hello world"
```

OO Design Patterns

Singleton

```
1 var Router = function() {  
2     // Singleton!  
3     if (Router.__instance__) {  
4         return Router.__instance__;  
5     }  
6  
7     Router.__instance__ = this;  
8     this.routes = {};  
9 };  
10  
11 Router.prototype.setRoute = function(match, fn) {  
12     this.routes[match] = fn;  
13 };  
14  
15 var myrouter = new Router();  
16 var another = new Router();  
17  
18 myrouter === another;
```

```
1 var Router = function() {
2     // Singleton!
3     if (Router.__instance__) {
4         return Router.__instance__;
5     }
6
7     function setRoute(match,fn) {
8         routes[match] = fn;
9     }
10
11    var routes = {};
12    var publicAPI = Router.__instance__ = {
13        setRoute: setRoute
14    };
15    return publicAPI;
16 };
17
18
19 var myrouter = new Router();
20 var another = new Router();
21
22 myrouter === another;
```

OO Design Patterns: singleton

Observer

```
1 function PageController(router) {  
2     this.router = router;  
3     this.router.on("navigate",  
4         this.fetchPage.bind(this)  
5     );  
6 }  
7 PageController.prototype.fetchPage = function(d) {  
8     $.ajax({  
9         url: d.page_url  
10    })  
11    .done(this.loaded.bind(this,d.page_url));  
12 };  
13 PageController.prototype.loaded = function(d,u) {  
14     // display the page content from `d`  
15     // ...  
16     this.router.emit("pageLoaded",u);  
17 };  
18  
19 var router = new Router();  
20 var thepage = new PageController(router);
```

prototype

Every single “object” is
built by a constructor call

prototype

A constructor makes an object “based on” its own **prototype**

prototype

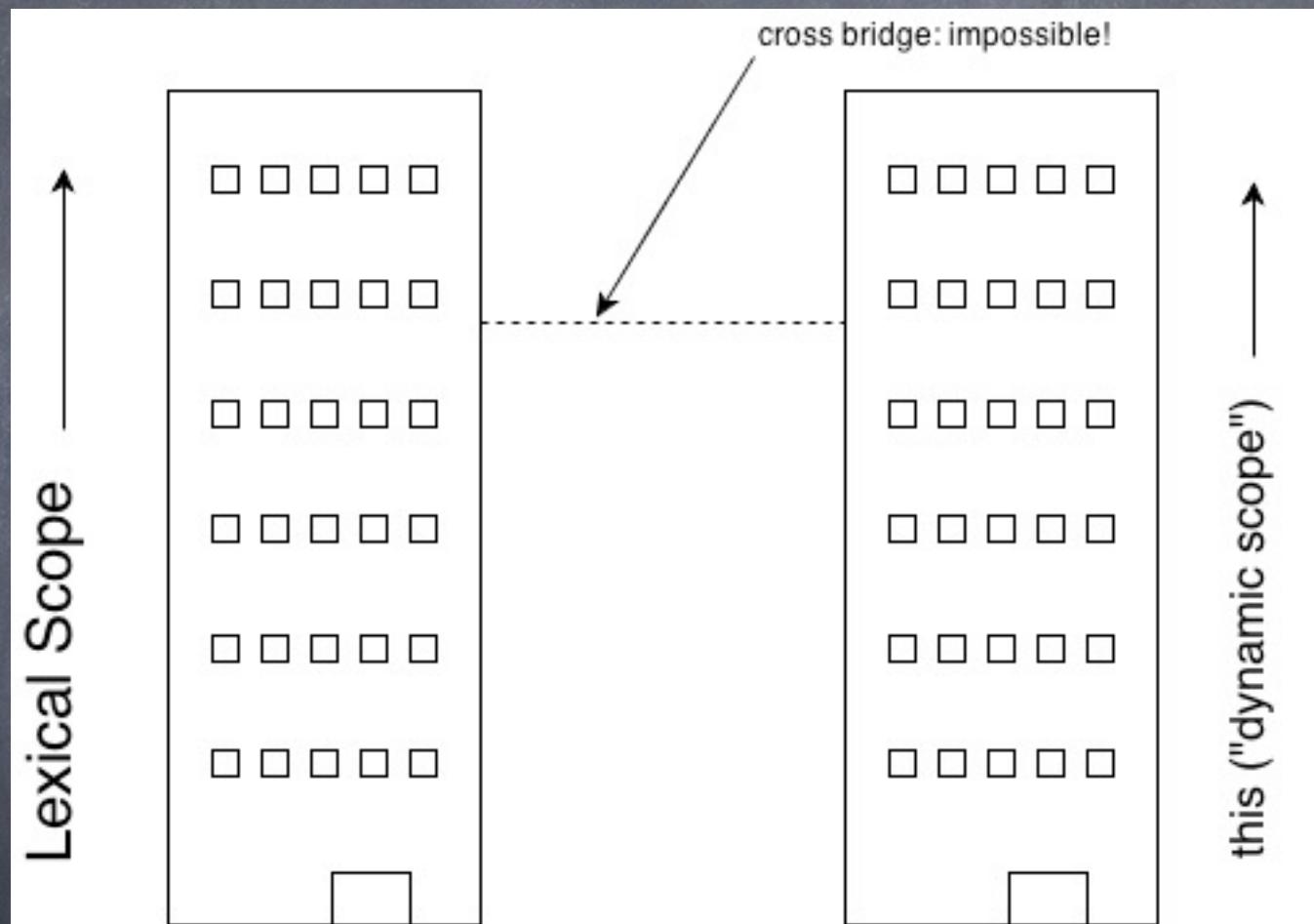
A constructor makes an
object **linked** to its own
prototype

prototype

```
1 function Foo(who) {  
2     this.me = who;  
3 }  
4 Foo.prototype.identify = function() {  
5     return "I am " + this.me;  
6 };  
7  
8 var a1 = new Foo("a1");  
9 var a2 = new Foo("a2");  
10  
11 a2.speak = function() {  
12     alert("Hello, " + this.identify() + ".");  
13 };  
14  
15 a1.constructor === Foo;  
16 a1.constructor === a2.constructor;  
17 a1.__proto__ === Foo.prototype;  
18 a1.__proto__ === a2.__proto__;
```

prototype

```
1 function Foo(who) {  
2     this.me = who;  
3 }  
4 Foo.prototype.identify = function() {  
5     return "I am " + this.me;  
6 };  
7  
8 var a1 = new Foo("a1");  
9 var a2 = new Foo("a2");  
10  
11 a2.speak = function() {  
12     alert("Hello, " + this.identify() + ".");  
13 };  
14  
15 a1.__proto__ === Object.getPrototypeOf(a1);  
16 a2.constructor === Foo;  
17 a1.__proto__ == a2.__proto__;  
18 a2.__proto__ == a2.constructor.prototype;
```



prototype

```
1 function Foo(who) {  
2     this.me = who;  
3 }  
4 Foo.prototype.identify = function() {  
5     return "I am " + this.me;  
6 };  
7  
8 var a1 = new Foo("a1");  
9 a1.identify(); // "I am a1"  
10  
11 a1.identify = function() { // <-- Shadowing  
12     alert("Hello, " +  
13         Foo.prototype.identify.call(this) +  
14         ".");  
15 };  
16  
17 a1.identify(); // alerts: "Hello, I am a1."
```

prototype

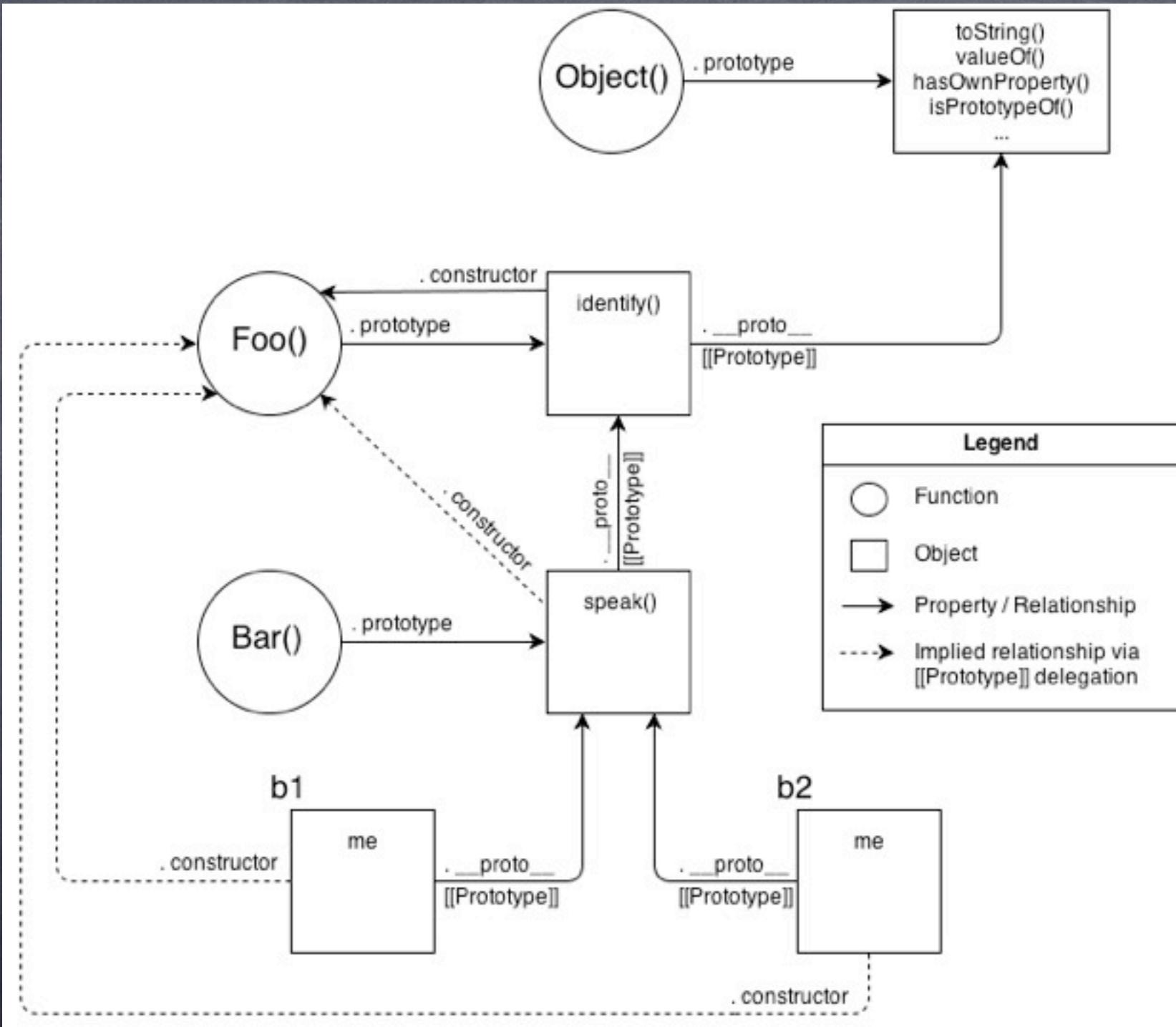
```
1 function Foo(who) {  
2     this.me = who;  
3 }  
4  
5 Foo.prototype.identify = function() {  
6     return "I am " + this.me;  
7 };  
8  
9 Foo.prototype.speak = function() {  
10    alert("Hello, " +  
11        this.identify() + // super unicorn magic  
12    ".");  
13 };  
14  
15 var a1 = new Foo("a1");  
16 a1.speak(); // alerts: "Hello, I am a1."
```

“Inheritance”

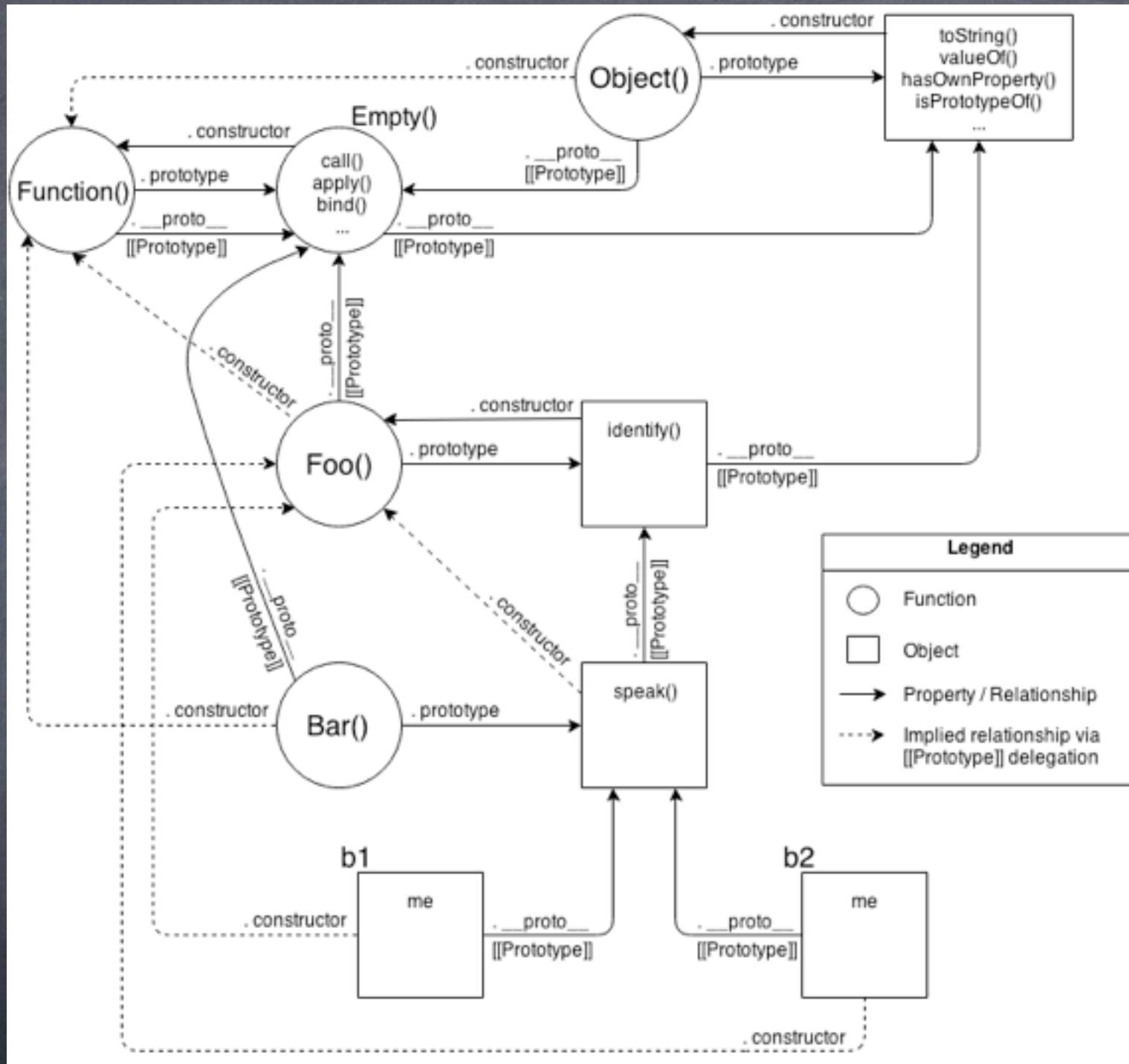
prototype

```
1 function Foo(who) {  
2     this.me = who;  
3 }  
4 Foo.prototype.identify = function() {  
5     return "I am " + this.me;  
6 };  
7  
8 function Bar(who) {  
9     Foo.call(this,who);  
10 }  
11 // Bar.prototype = new Foo(); // Or...  
12 Bar.prototype = Object.create(Foo.prototype);  
13 // NOTE: .constructor is borked here, need to fix  
14  
15 Bar.prototype.speak = function() {  
16     alert("Hello, " + this.identify() + ".");  
17 };  
18  
19 var b1 = new Bar("b1");  
20 var b2 = new Bar("b2");  
21  
22 b1.speak(); // alerts: "Hello, I am b1."  
23 b2.speak(); // alerts: "Hello, I am b2."
```

prototype: objects linked



prototype: objects linked



prototype: objects linked

Quiz

1. What is a constructor call?
2. What is [[Prototype]] and where does it come from?
3. How does [[Prototype]] affect the behavior of an object?
4. How do we find out where an object's [[Prototype]] points to (3 ways)?

prototype

remember how **this** has a
pesky way of getting
unassigned?

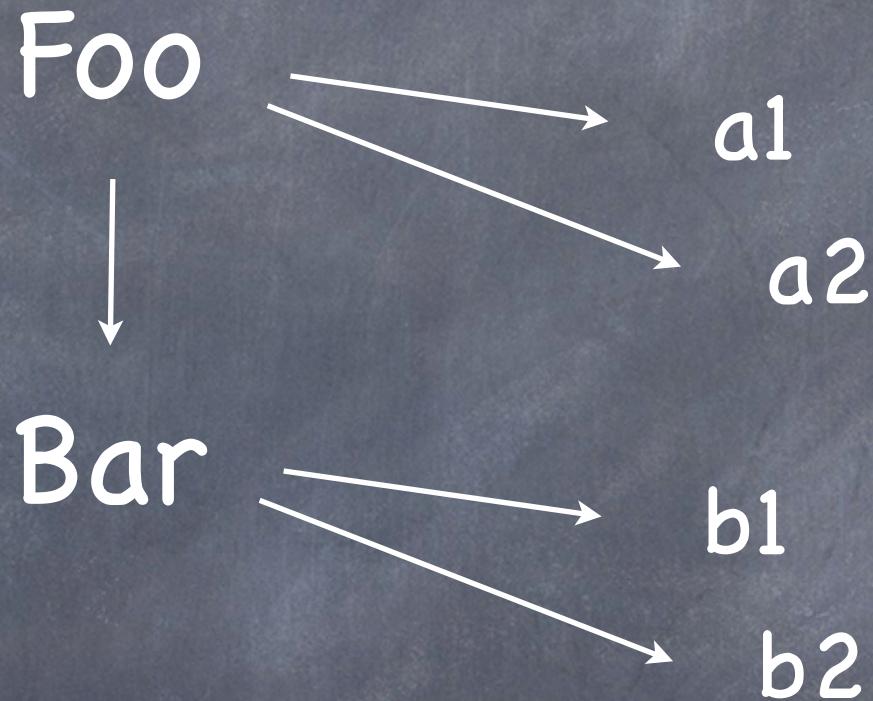
prototype: this redux

```
1 function Foo(who) {  
2     this.me = who;  
3 }  
4  
5 Foo.prototype.speak = function() {  
6     alert("Hello, I am " + this.me + ".");  
7 };  
8  
9 var a1 = new Foo("a1");  
10  
11 $("#speak").click(a1.speak);
```

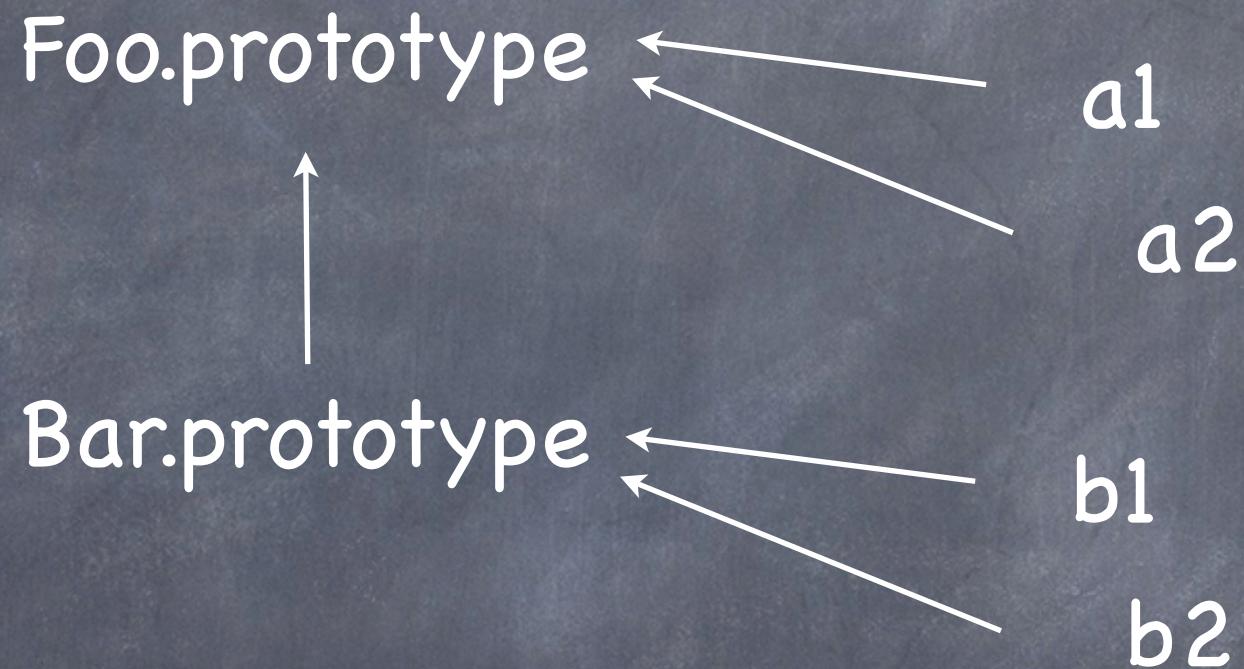
prototype: this redux

(exercise #2: 20min)

Inheritance



OO: **classical inheritance**



(another design pattern)

OO: “prototypal inheritance”

JS “Inheritance” “Behavior Delegation”

```
1 function Foo(who) {  
2     this.me = who;  
3 }  
4  
5 Foo.prototype.speak = function() {  
6     alert("Hello, I am " + this.me + ".");  
7 };  
8  
9 var a1 = new Foo("a1");  
10  
11 a1.speak(); // alerts: "Hello, I am a1."
```

OO: inheritance delegation

Let's simplify!

OO: behavior delegation

OLOO:
Objects Linked to Other
Objects

OLOO

```
1 function Foo(who) {  
2     this.me = who;  
3 }  
4 Foo.prototype.identify = function() {  
5     return "I am " + this.me;  
6 };  
7  
8 function Bar(who) {  
9     Foo.call(this,who);  
10 }  
11 Bar.prototype = Object.create(Foo.prototype);  
12  
13 Bar.prototype.speak = function() {  
14     alert("Hello, " + this.identify() + ".");  
15 };  
16  
17 var b1 = new Bar("b1");  
18 b1.speak(); // alerts: "Hello, I am b1."
```

OLOO: delegated objects

```
1 function Foo(who) {  
2     this.me = who;  
3 }  
4 Foo.prototype.identify = function() {  
5     return "I am " + this.me;  
6 };  
7  
8 function Bar(who) {  
9     Foo.call(this,who);  
10 }  
11 Bar.prototype = Object.create(Foo.prototype);  
12  
13 Bar.prototype.speak = function() {  
14     alert("Hello, " + this.identify() + ".");  
15 };  
16  
17 var b1 = Object.create(Bar.prototype);  
18 Bar.call(b1,"b1");  
19 b1.speak();
```

OLOO: delegated objects

```
1 function Foo(who) {  
2     this.me = who;  
3 }  
4 Foo.prototype.identify = function() {  
5     return "I am " + this.me;  
6 };  
7  
8 var Bar = Object.create(Foo.prototype);  
9 Bar.init = function(who) {  
10     Foo.call(this, who);  
11 };  
12 Bar.speak = function() {  
13     alert("Hello, " + this.identify() + ".");  
14 };  
15  
16 var b1 = Object.create(Bar);  
17 b1.init("b1");  
18 b1.speak(); // alerts: "Hello, I am b1."
```

OLOO: delegated objects

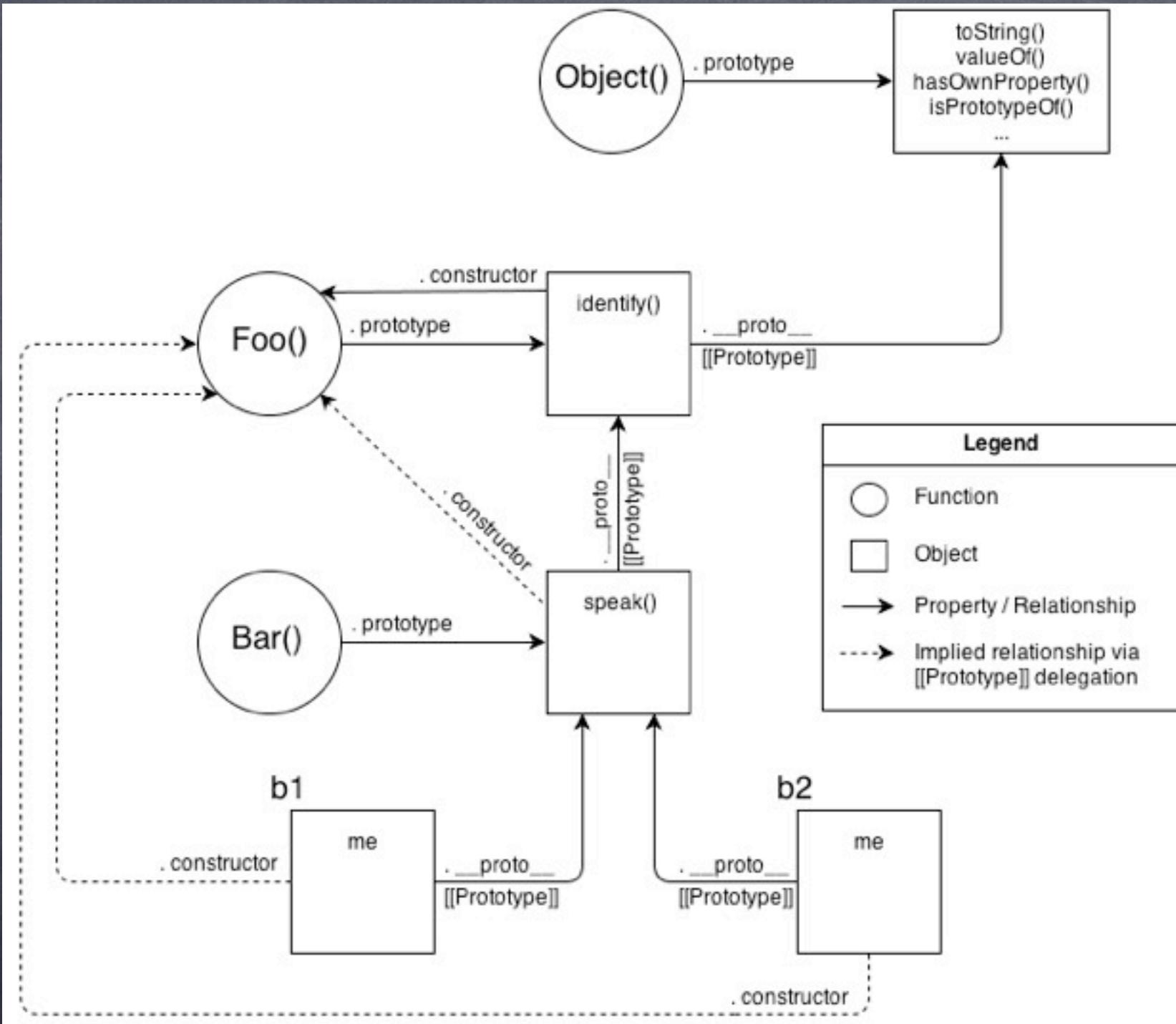
```
1 var Foo = {  
2     init: function(who) {  
3         this.me = who;  
4     },  
5     identify: function() {  
6         return "I am " + this.me;  
7     }  
8 };  
9  
10 var Bar = Object.create(Foo);  
11  
12 Bar.speak = function() {  
13     alert("Hello, " + this.identify() + ".");  
14 };  
15  
16 var b1 = Object.create(Bar);  
17 b1.init("b1");  
18 b1.speak(); // alerts: "Hello, I am b1."
```

OLOO: delegated objects

Mental Models

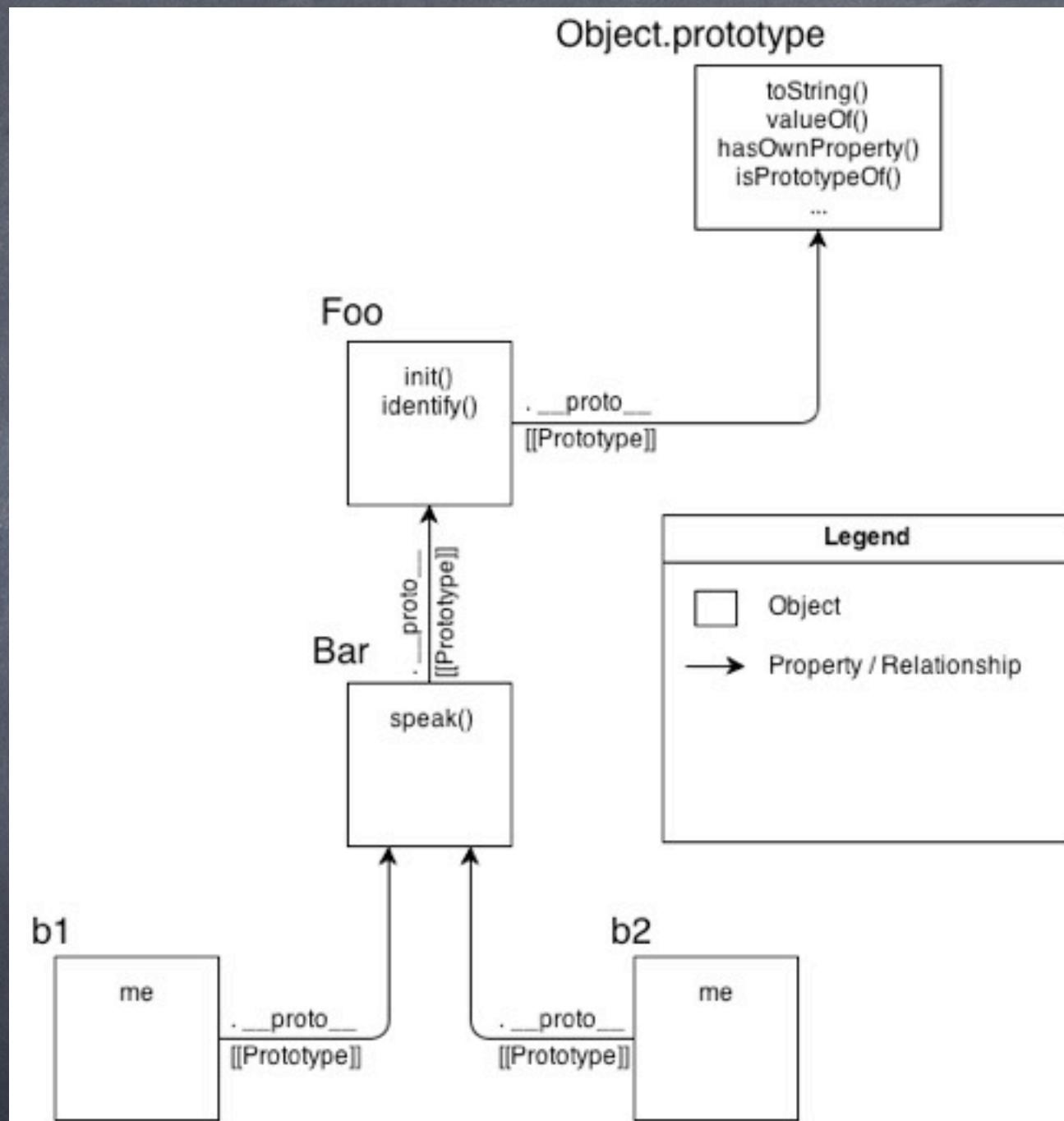
OO vs. OLOO

```
1 function Foo(who) {
2     this.me = who;
3 }
4 Foo.prototype.identify = function() {
5     return "I am " + this.me;
6 };
7
8 function Bar(who) {
9     Foo.call(this,who);
10}
11 // Bar.prototype = new Foo(); // Or...
12 Bar.prototype = Object.create(Foo.prototype);
13 // NOTE: .constructor is borked here, need to fix
14
15 Bar.prototype.speak = function() {
16     alert("Hello, " + this.identify() + ".");
17 };
18
19 var b1 = new Bar("b1");
20 var b2 = new Bar("b2");
21
22 b1.speak(); // alerts: "Hello, I am b1."
23 b2.speak(); // alerts: "Hello, I am b2."
```



OO: old & busted

```
1 var Foo = {  
2     init: function(who) {  
3         this.me = who;  
4     },  
5     identify: function() {  
6         return "I am " + this.me;  
7     }  
8 };  
9  
10 var Bar = Object.create(Foo);  
11  
12 Bar.speak = function() {  
13     alert("Hello, " + this.identify() + ".");  
14 };  
15  
16 var b1 = Object.create(Bar);  
17 b1.init("b1");  
18 var b2 = Object.create(Bar);  
19 b2.init("b2");  
20  
21 b1.speak(); // alerts: "Hello, I am b1."  
22 b2.speak(); // alerts: "Hello, I am b2."
```



OLOO: new hotness

```
1 if (!Object.create) {  
2     Object.create = function (o) {  
3         function F() {}  
4         F.prototype = o;  
5         return new F();  
6     };  
7 }
```

OLOO: Object.create()

Delegation Design Pattern

Delegation-Oriented Design

Parent-Child

Peer-Peer

Delegation-Oriented Design

Quiz

1. How is JavaScript's `[[Prototype]]` chain not like traditional/classical inheritance?
2. What does `[[Prototype]]` "delegation" mean and how does it describe object linking in JS?
3. What are the benefits of the "behavior delegation" design pattern? What are the tradeoffs of using `[[Prototype]]`?

(exercise #3: 10min)

Kyle Simpson
@getify
<http://getify.me>

Thanks!

Questions?