



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

- LABjs
- grips
- asynquence

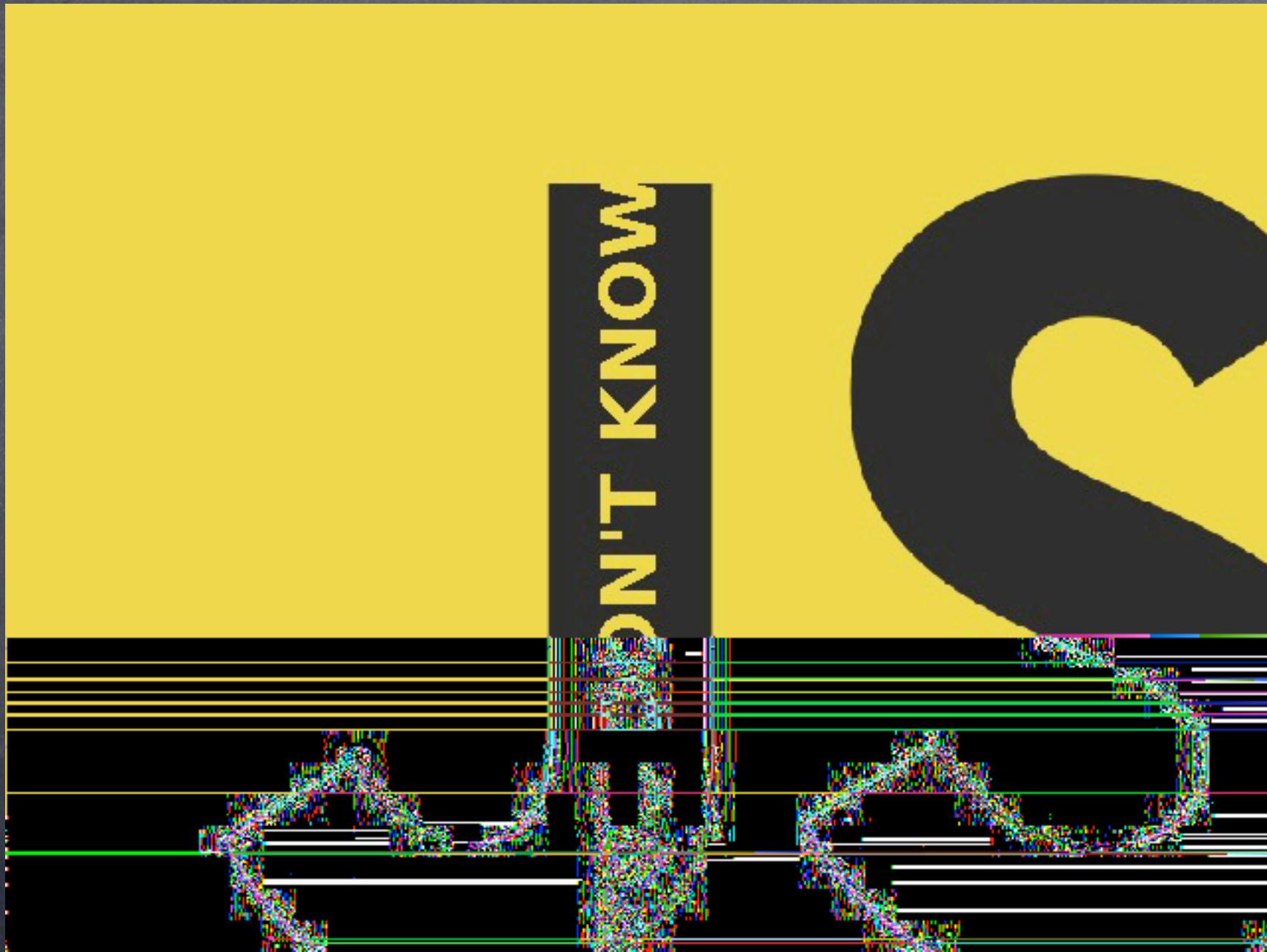
# Kyle Simpson

## @getify

### <http://getify.me>

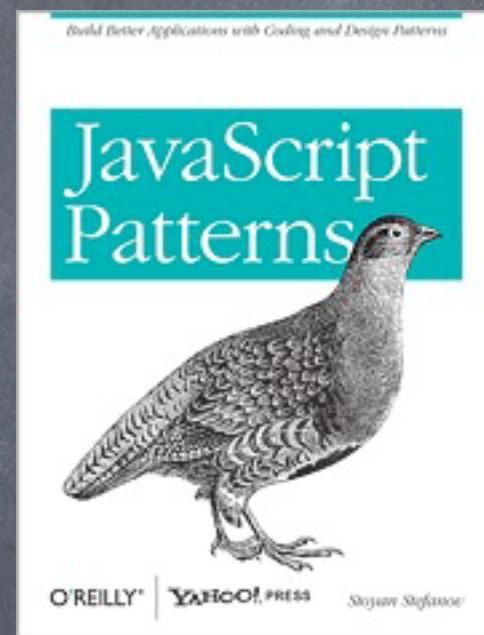


<http://speakerdeck.com/getify>



<http://YouDontKnowJS.com>





<https://developer.mozilla.org/en-US/docs/JavaScript>

<https://github.com/rwldrn/idiomatic.js>

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

<http://wiki.ecmascript.org/doku.php?id=harmony:proposals>

# Advanced JavaScript

## The “What you need to know” Parts



# Agenda

## Scope, Closures

- Nested Scope
- Hoisting
- this
- 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!
```

```
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?

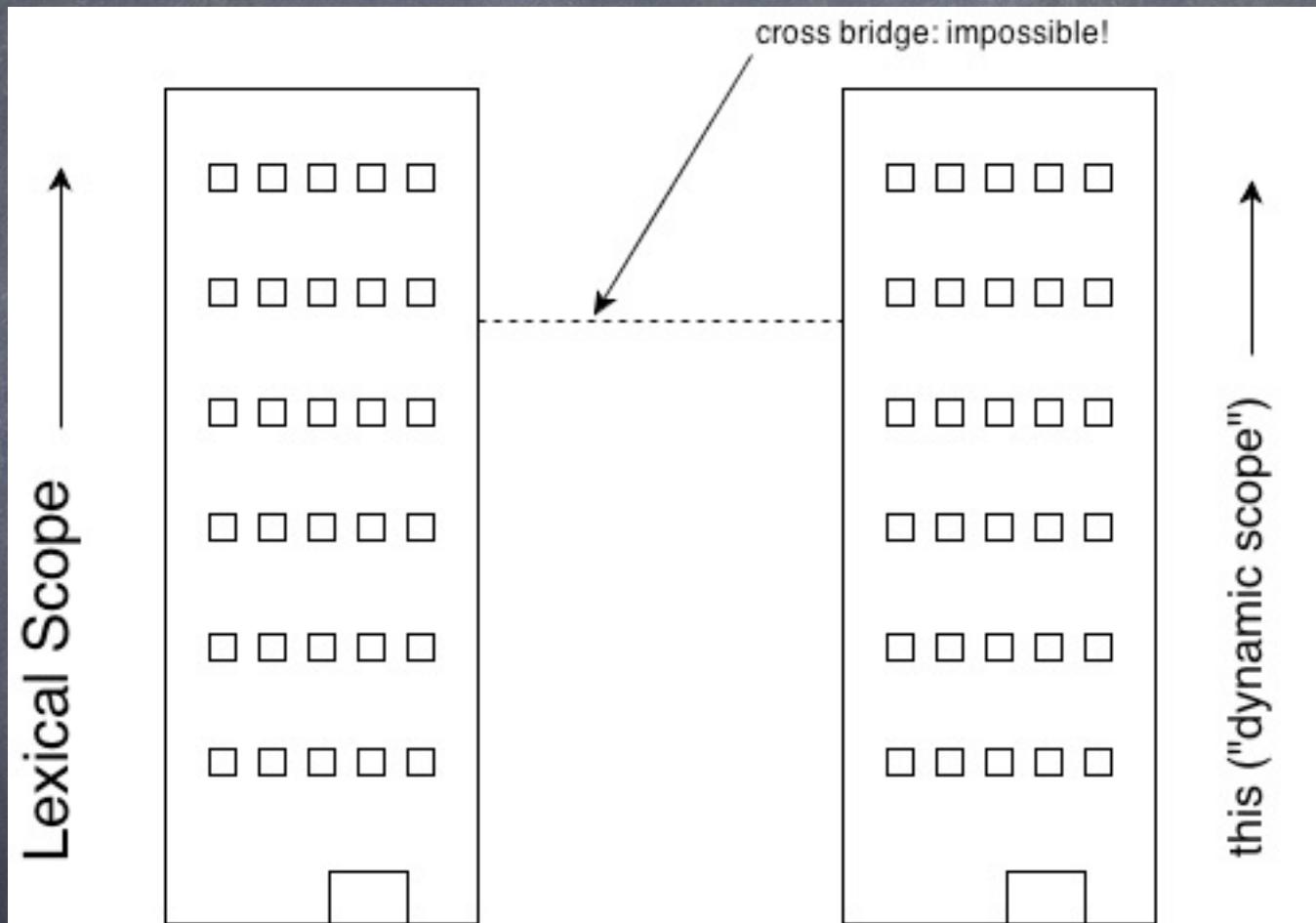
```
1 var foo;  
2  
3 try {  
4     foo.length;  
5 }  
6 catch (err) {  
7     console.log(err); // TypeError  
8 }  
9  
10 console.log(err); // ReferenceError
```

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     d = b - a;  
13     d = 3; // ??  
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

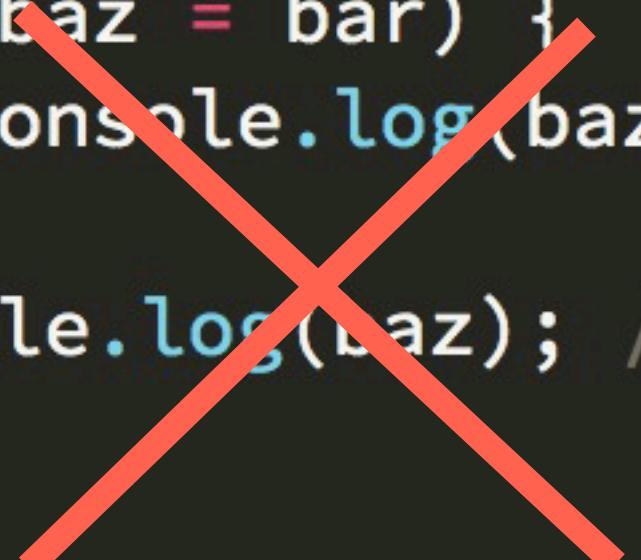
```
1 function foo() {  
2     var bar = "bar";  
3     for (let i=0; i<bar.length; i++) {  
4         console.log(bar.charAt(i));  
5     }  
6     console.log(i); // ReferenceError  
7 }  
8  
9 foo();
```

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

```
1 let (foo) {  
2     foo = "foo";  
3     console.log(foo); // "foo"  
4 }  
5  
6 foo; // ReferenceError
```

<https://github.com/getify/let-er>

Block Scope: let

```
1 try{throw void 0}catch
2 /*let*/(foo) {
3   foo = "foo";
4   console.log(foo);
5 }
6
7 foo; // Reference Error!
```

<https://github.com/getify/let-er>

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 };
```

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

Hoisting: functions first

```
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

(exercise #1: 10min)

this

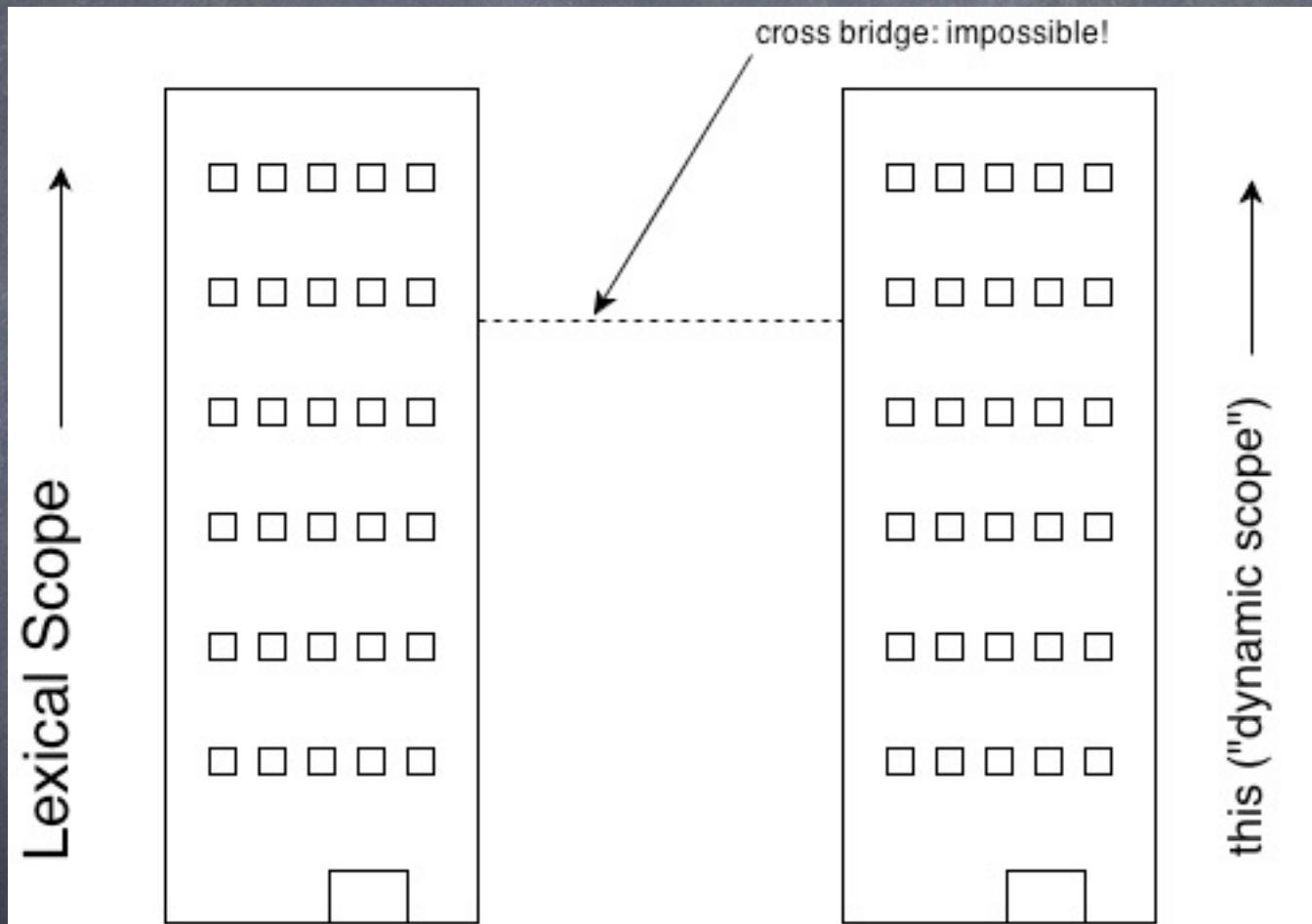
Every function, while executing, has a reference to its current execution context, called **this**.

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     var bar = "bar1";  
3     baz();  
4 }  
5 function baz() {  
6     console.log(this.bar);  
7 }  
8  
9 var bar = "bar2";  
10 foo();           // ???
```

this: binding confusion

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

this: binding confusion

```
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

```
1 if (!Function.prototype.bind2) {  
2     Function.prototype.bind2 =  
3         function(o) {  
4             var fn = this; // the function!  
5             return function() {  
6                 return fn.apply(o, arguments);  
7             };  
8         };  
9     }  
10  
11 function foo(baz) {  
12     console.log(this.bar + " " + baz);  
13 }  
14  
15 var obj = { bar: "bar" };  
16 foo = foo.bind2(obj);  
17  
18 foo("baz");           // "bar baz"
```

this: hard binding

```
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"  
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"
```

this: new, 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 seal a specific **this** to a function? Why do that? Why not?
5. How do you create a new **this**?

this

Closure

Remember lexical scope?

Closure absolutely depends on it. Go review it if you're still confused. :)

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 bar(); // "bar"
3
4 module foo from "foo";
5 foo.bar(); // "bar"
```

Closure: future/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 #2: 20min)



# Agenda

## Object-Orienting

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

OO Design Patterns

# Singleton

```
1 var Router = function() {  
2     // Singleton!  
3     if (Router.__instance__) {  
4         return Router.__instance__;  
5     }  
6     Router.__instance__ = this;  
7     this.routes = {};  
8 };  
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;
```

# 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  
function

prototype

Each time a constructor is  
called, a new object is  
created

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__;
```

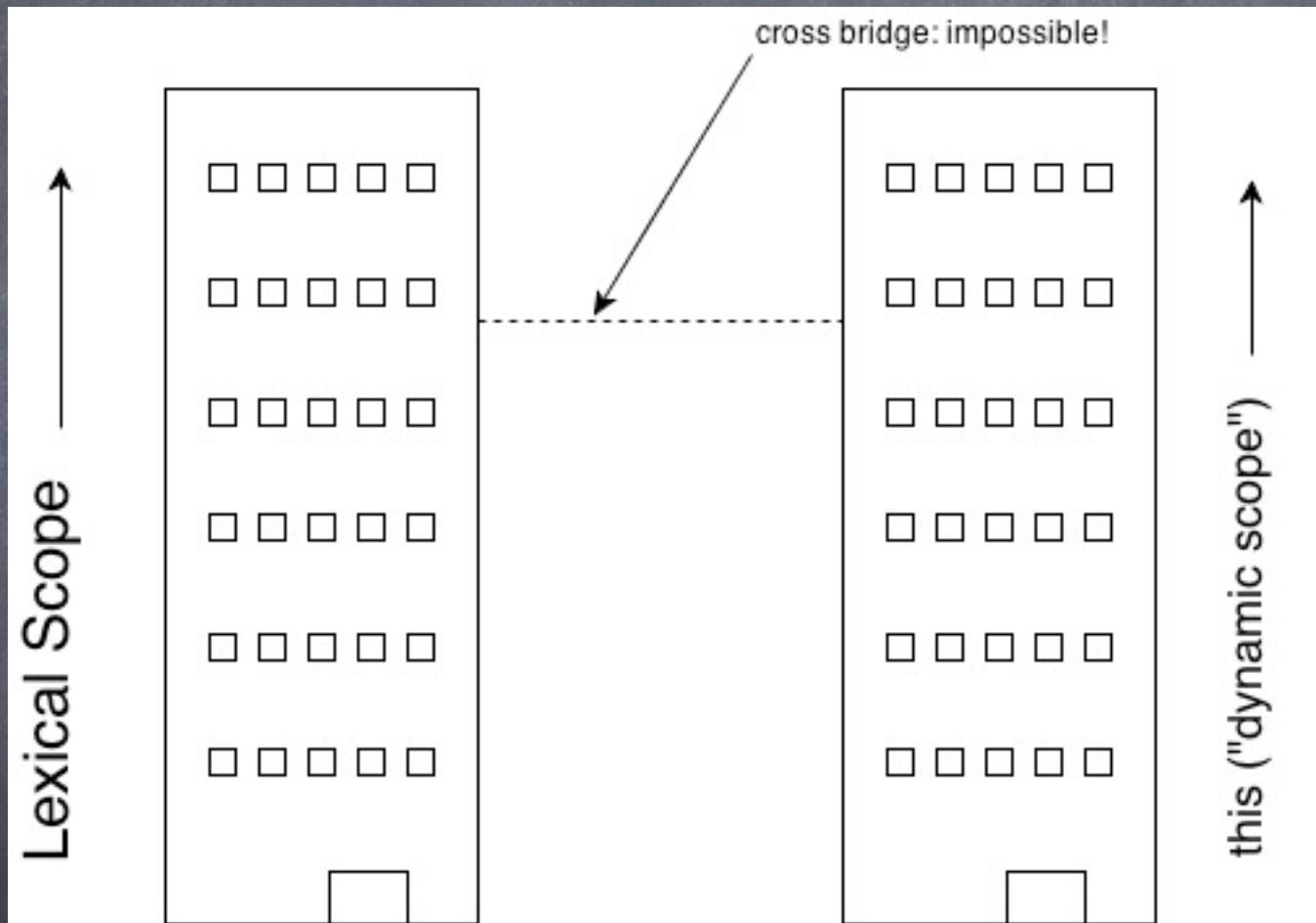
prototype

```
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;
```

```
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."
```

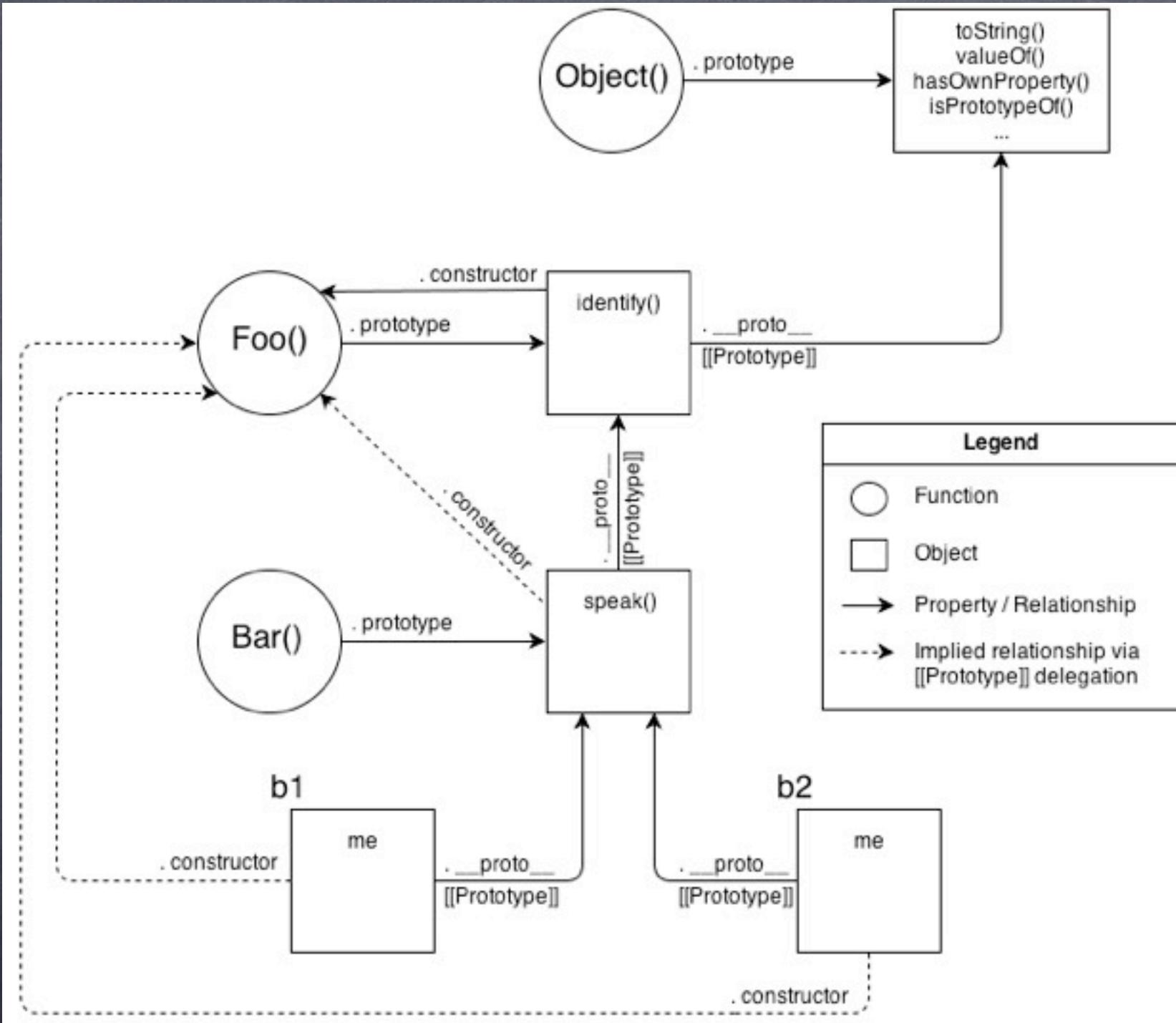
prototype

```
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."
```

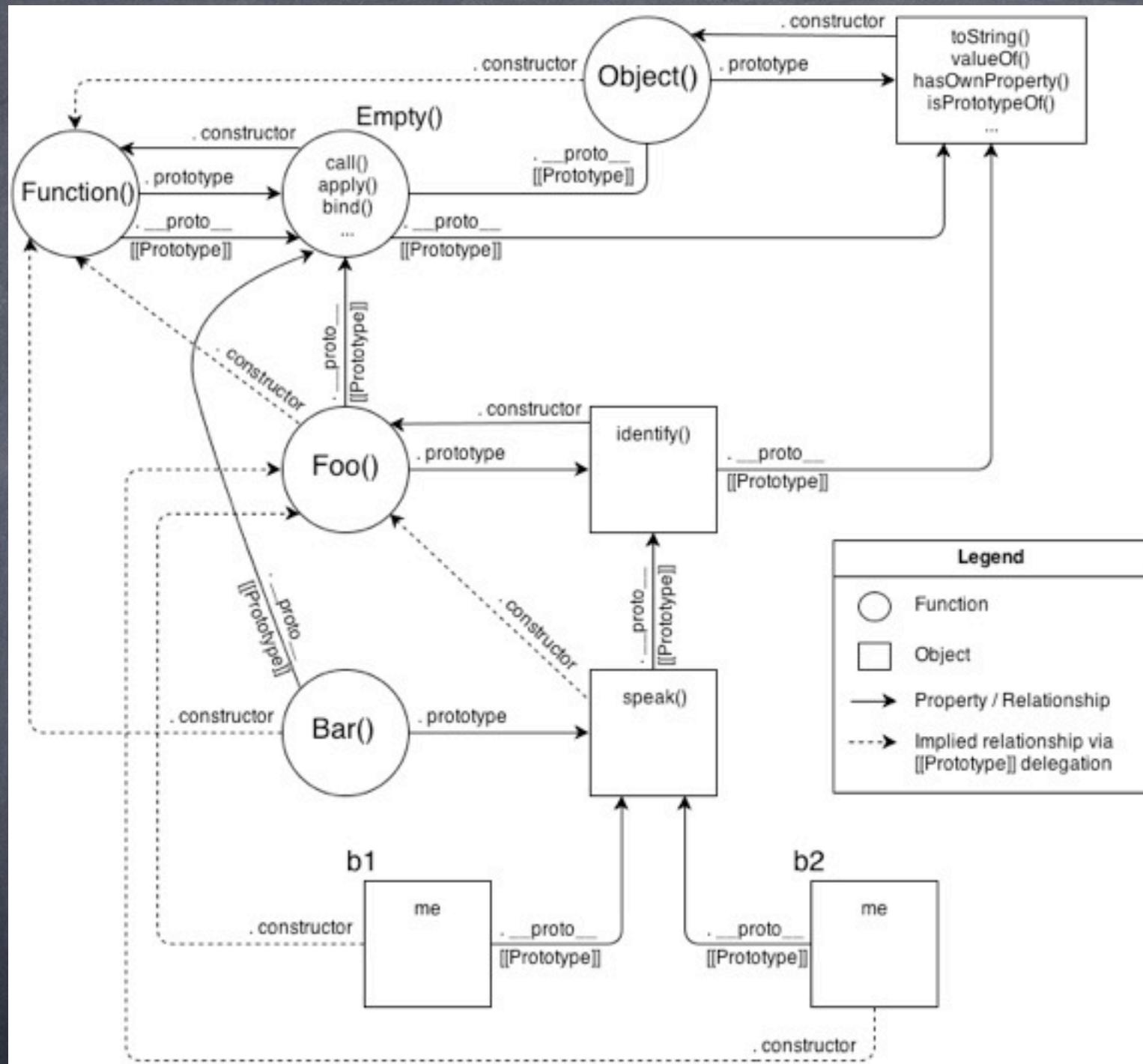


```
1  function Foo(who) {  
2      this.me = who;  
3  }  
4  Foo.prototype.identify = function() {  
5      return "I am " + this.me;  
6  };  
7  
8  function Bar(who) {  
9      Foo.call(this,who);  
10 }  
11 // Bar.prototype = new Foo(); // Or...  
12 Bar.prototype = Object.create(Foo.prototype);  
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?
2. What is a [[Prototype]] and where does it come from?
3. How does a [[Prototype]] affect 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?

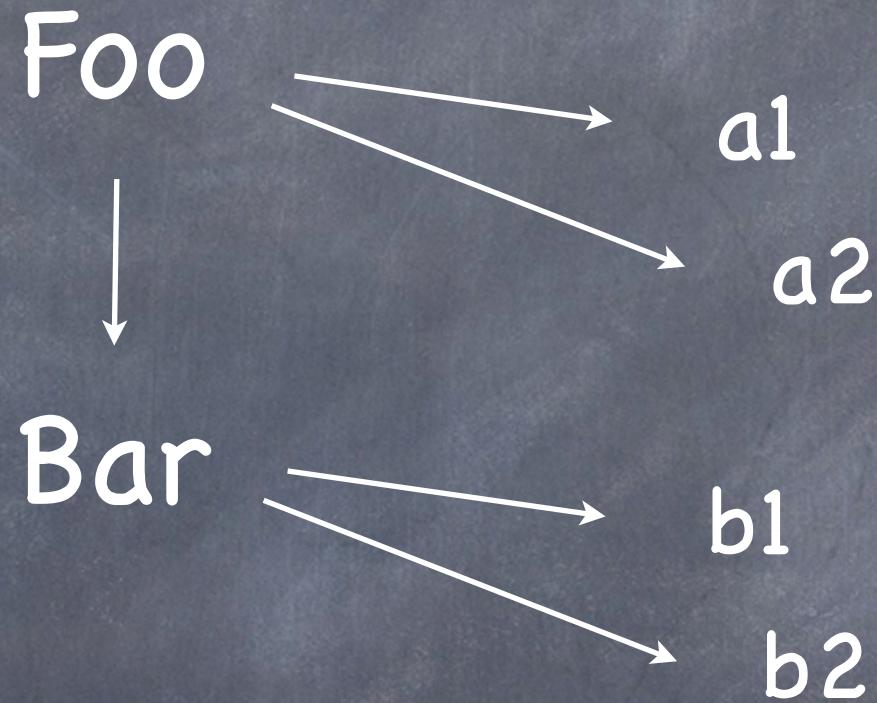
prototype: this redux

```
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);
```

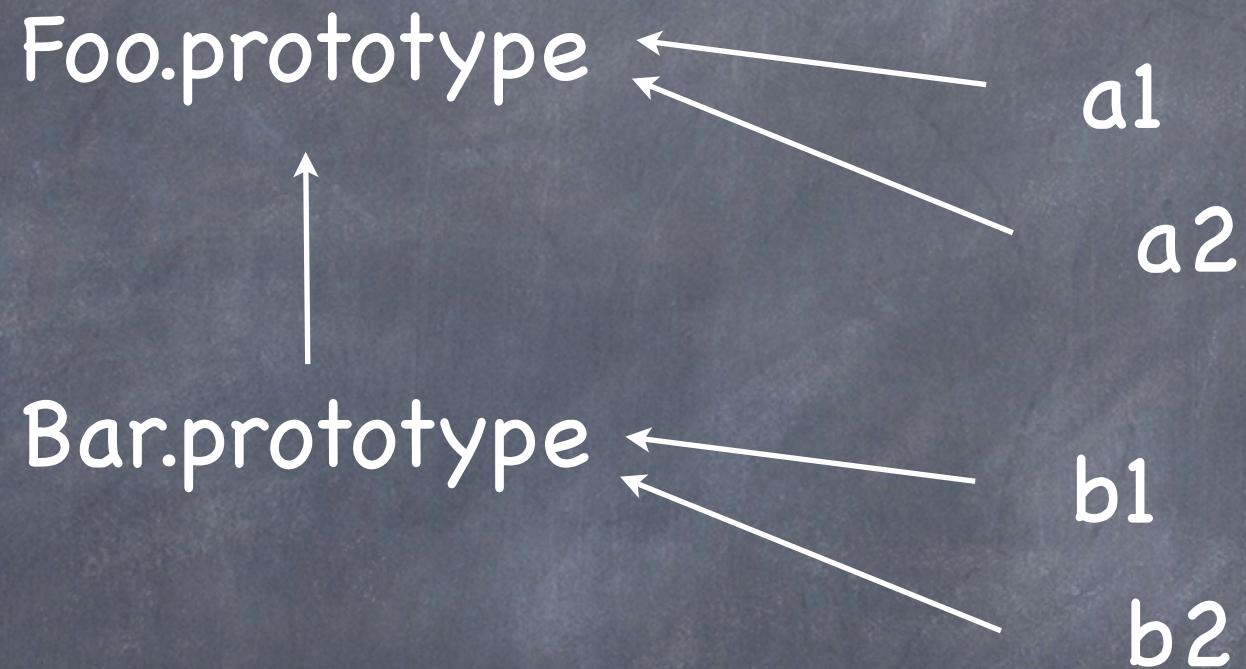
prototype: this redux

(exercise #3: 10min)

# 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

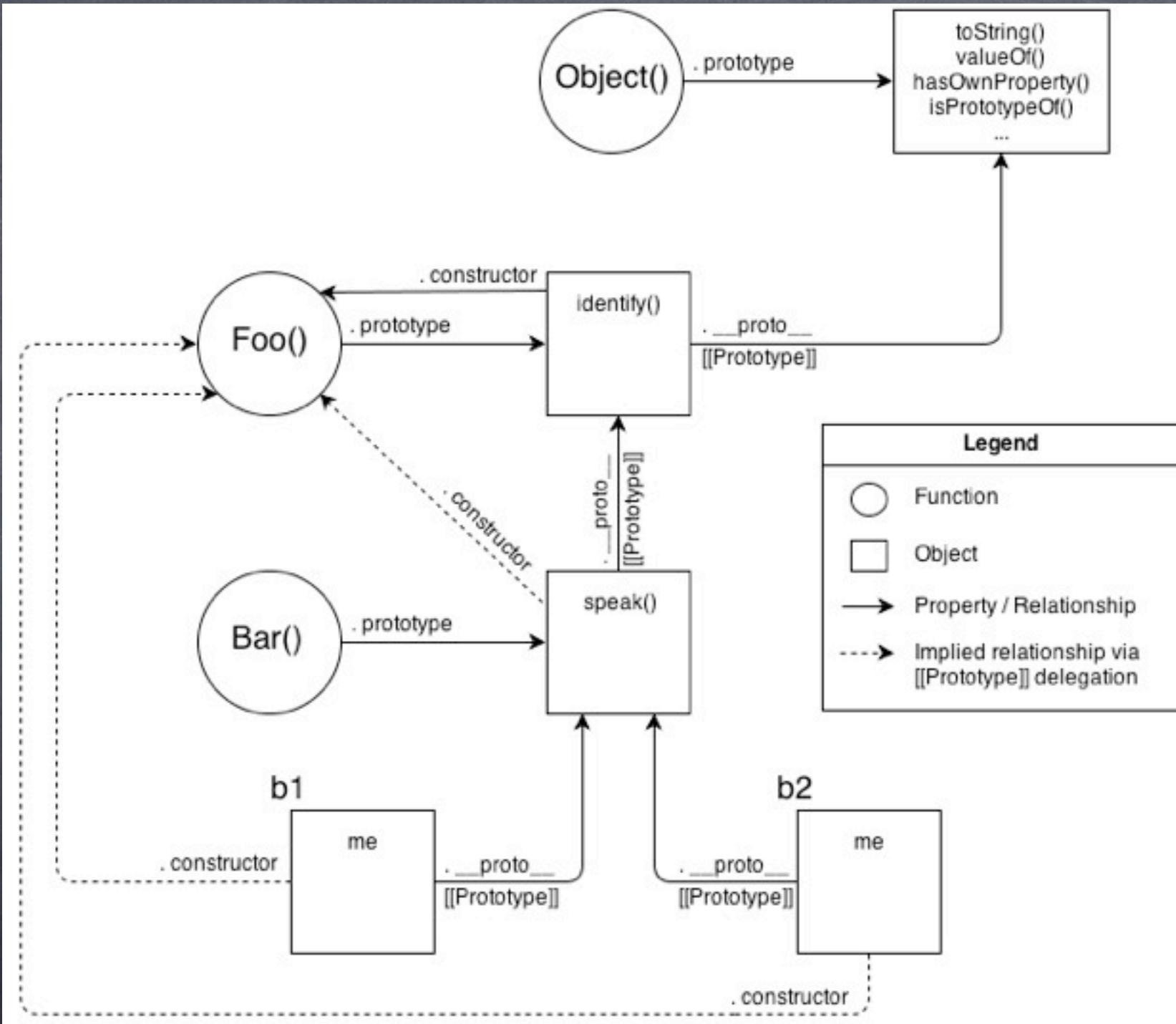
```
1 function Foo(who) {  
2     this.me = who;  
3 }  
4 Foo.prototype.identify = function() {  
5     return "I am " + this.me;  
6 };  
7  
8 var Bar = Object.create(Foo.prototype);  
9 Bar.init = function(who) {  
10     Foo.call(this, who);  
11 };  
12 Bar.speak = function() {  
13     alert("Hello,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

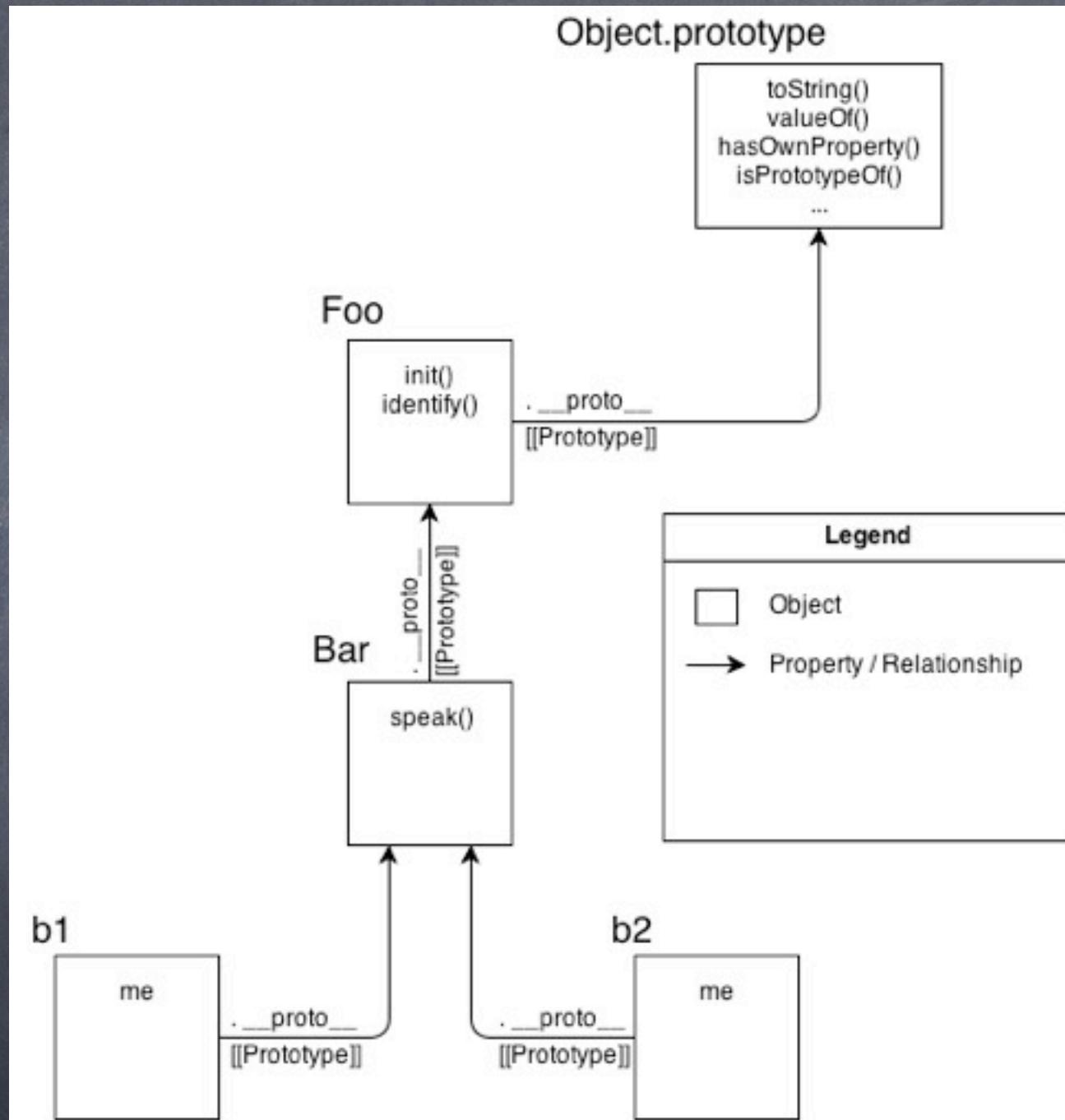
OLOO: delegated objects

```
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()

<https://gist.github.com/getify/5572383>

<https://gist.github.com/getify/5226305>

OLOO: object creation

# Quiz

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

(exercise #4: 15min)



# Agenda

## Async Patterns

- Callbacks
- Generators/Coroutines
- Promises

# Callbacks

Async Patterns

```
1 setTimeout(function(){
2     console.log("callback!");
3 },1000);
```

Async Patterns: callbacks

```
1 setTimeout(function(){
2     console.log("one");
3     setTimeout(function(){
4         console.log("two");
5         setTimeout(function(){
6             console.log("three");
7             },1000);
8         },1000);
9     },1000);
```

Async Patterns: “callback hell”

```
1 function one(cb) {  
2     console.log("one");  
3     setTimeout(cb,1000);  
4 }  
5 function two(cb) {  
6     console.log("two");  
7     setTimeout(cb,1000);  
8 }  
9 function three() {  
10    console.log("three");  
11 }  
12  
13 one(function(){  
14     two(three);  
15 }));
```

Async Patterns: “callback hell”

# Inversion of Control

Async Patterns: “callback hell”

```
1 function trySomething(ok, err) {  
2     setTimeout(function(){  
3         var num = Math.random();  
4         if (num > 0.5) ok(num);  
5         else err(num);  
6     }, 1000);  
7 }  
8  
9 trySomething(  
10     function(num){  
11         console.log("Success: " + num);  
12     },  
13     function(num){  
14         console.log("Sorry: " + num);  
15     }  
16 );
```

Async Patterns: separate callbacks

```
1 function trySomething(cb) {  
2     setTimeout(function(){  
3         var num = Math.random();  
4         if (num > 0.5) cb(null,num);  
5         else cb("Too low!");  
6     },1000);  
7 }  
8  
9 trySomething(function(err,num){  
10    if (err) {  
11        console.log(err);  
12    }  
13    else {  
14        console.log("Number: " + num)  
15    }  
16});
```

Async Patterns: “error-first style”

```
1 function getData(d,cb) {  
2     setTimeout(function(){ cb(d); },1000);  
3 }  
4  
5 getData(10,function(num1){  
6     var x = 1 + num1;  
7     getData(30,function(num2){  
8         var y = 1 + num2;  
9         getData(  
10            "Meaning of life: " + (x + y),  
11            function(answer){  
12                console.log(answer);  
13                // Meaning of life: 42  
14            }  
15        );  
16    });  
17});
```

## Async Patterns: nested-callback tasks

# Generators (`yield`)

```
1 function* gen() {  
2     console.log("Hello");  
3     yield null;  
4     console.log("World");  
5 }  
6  
7 var it = gen();  
8 it.next(); // prints "Hello"  
9 it.next(); // prints "World"
```

```
1 function coroutine(g) {  
2     var it = g();  
3     return function(){  
4         return it.next.apply(it,arguments);  
5     };  
6 }
```

Async Patterns: generator coroutines

```
1 var run = coroutine(function*(){
2     var x = 1 + (yield null);
3     var y = 1 + (yield null);
4     yield (x + y);
5 });
6
7 run();
8 run(10);
9 console.log(
10    "Meaning of life: " + run(30).value
11 );
```

Async Patterns: generator messages

```
1 function getData(d) {  
2     setTimeout(function(){run(d); },1000);  
3 }  
4  
5 var run = coroutine(function*(){  
6     var x = 1 + (yield getData(10));  
7     var y = 1 + (yield getData(30));  
8     var answer = (yield getData(  
9         "Meaning of life: " + (x + y)  
10    ));  
11    console.log(answer);  
12    // Meaning of life: 42  
13});  
14  
15 run();
```

Promises  
“continuation events”

```
1 var wait = jQuery.Deferred();
2 var p = wait.promise();
3
4 p.done(function(value){
5     console.log(value);
6 });
7
8 setTimeout(function(){
9     wait.resolve(Math.random());
10 },1000);
```

Async Patterns: jQuery-style promises

```
1 function waitForN(n) {  
2     var d = $.Deferred();  
3     setTimeout(d.resolve,n);  
4     return d.promise();  
5 }  
6  
7 waitForN(1000)  
8 .then(function(){  
9     console.log("Hello world");  
10    return waitForN(2000);  
11 })  
12 .then(function(){  
13     console.log("finally!")  
14 });
```

Async Patterns: jQuery-style promises

```
1 function getData(d) {  
2     return new Promise(function(resolve, reject){  
3         setTimeout(function(){resolve(d); },1000);  
4     });  
5 }  
6  
7 var x;  
8  
9 getData(10)  
10 .then(function(num1){  
11     x = 1 + num1;  
12     return getData(30);  
13 })  
14 .then(function(num2){  
15     var y = 1 + num2;  
16     return getData("Meaning of life: " +(x + y));  
17 })  
18 .then(function(answer){  
19     console.log(answer);  
20     // Meaning of life: 42  
21 });
```

# Async Patterns: (native) promise tasks

sequence = automatically  
chained promises

Async Patterns: promises sequence

```
1 ASQ()
2 .then(function(done){
3     setTimeout(done,1000);
4 })
5 .gate(
6     function(done){
7         setTimeout(done,1000);
8     },
9     function(){
10         setTimeout(done,1000);
11     }
12 )
13 .then(function(){
14     console.log("2 seconds passed!");
15 }));
```

Async Patterns: sequences & gates

```
1 function getData(d) {  
2     return ASQ(function(done){  
3         setTimeout(function(){done(d); },1000);  
4     });  
5 }  
6  
7 ASQ()  
8 .waterfall(  
9     function(done){ getData(10).pipe(done); },  
10    function(done){ getData(30).pipe(done); }  
11 )  
12 .seq(function(num1,num2){  
13     var x = 1 + num1;  
14     var y = 1 + num2;  
15     return getData("Meaning of life: " + (x + y));  
16 })  
17 .val(function(answer){  
18     console.log(answer);  
19     // Meaning of life: 42  
20 });
```

## Async Patterns: sequence tasks

```
1 function getData(d) {
2     return ASQ(function(done){
3         setTimeout(function(){done(d); },1000);
4     });
5 }
6
7 ASQ()
8 .runner(function*(){
9     var x = 1 + (yield getData(10));
10    var y = 1 + (yield getData(30));
11    var answer = yield (getData(
12        "Meaning of life: " + (x + y)
13    ));
14    return answer;
15 })
16 .val(function(answer){
17     console.log(answer);
18     // Meaning of life: 42
19 }));
```

Async Patterns: generator+sequence tasks

```
2 .runner(  
3     // v1 will be set as `2` here  
4     function*(v1) {  
5         console.log("initial v1:" + v1); // initial v1:2  
6         v1 += yield null;  
7         console.log("now v1:" + v1); // now v1:12  
8         v1 += yield getData(v1 * 2);  
9         console.log("finally v1:" + v1); // finally v1:84  
10        yield getData(v1 + 1);  
11    },  
12    function*() {  
13        var v2 = yield getData(10);  
14        console.log("initial v2:" + v2); // initial v2:24  
15        v2 += yield getData(v2 * 3);  
16        console.log("finally v2:" + v2); // finally v2:109  
17        yield getData(v2 + 6);  
18    }  
19 )  
20 .val(function(msg){  
21     console.log("Result: " + msg); // Result: 115  
22});
```

# Quiz

1. What is “callback hell”? Why do callbacks suffer from “inversion of control”?
2. How do you pause a generator? How do you resume it?
3. What is a Promise? How does it solve inversion of control issues?
4. How do we combine generators and promises for flow control?

(exercise #5: 15min)



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

Thanks!

Questions?