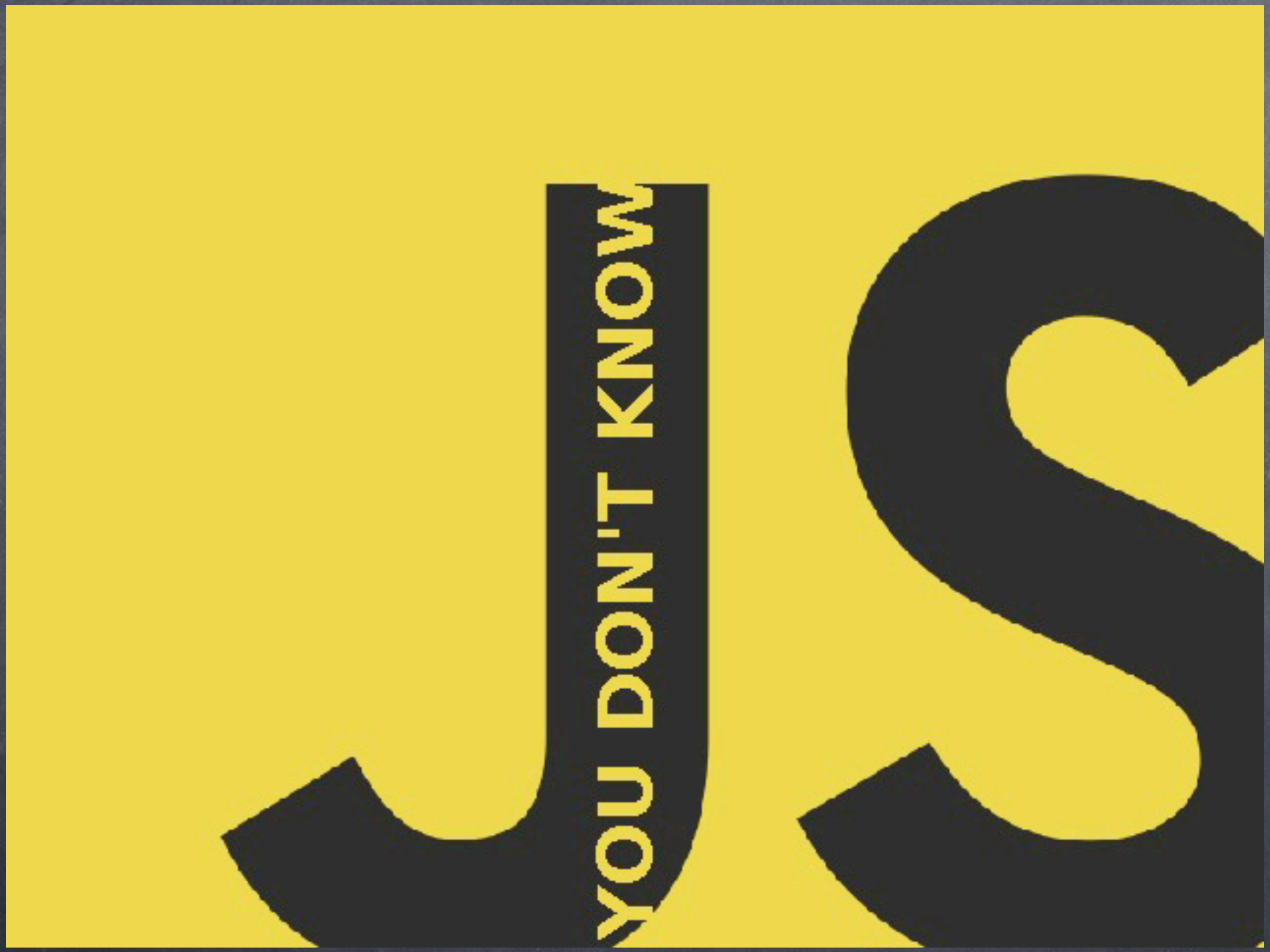
Kyle Simpson
@getify
http://getify.me

- LABjs
- grips
- asynquence

Kyle Simpson
@getify
http://getify.me

http://speakerdeck.com/getify



http://YouDontKnowJS.com

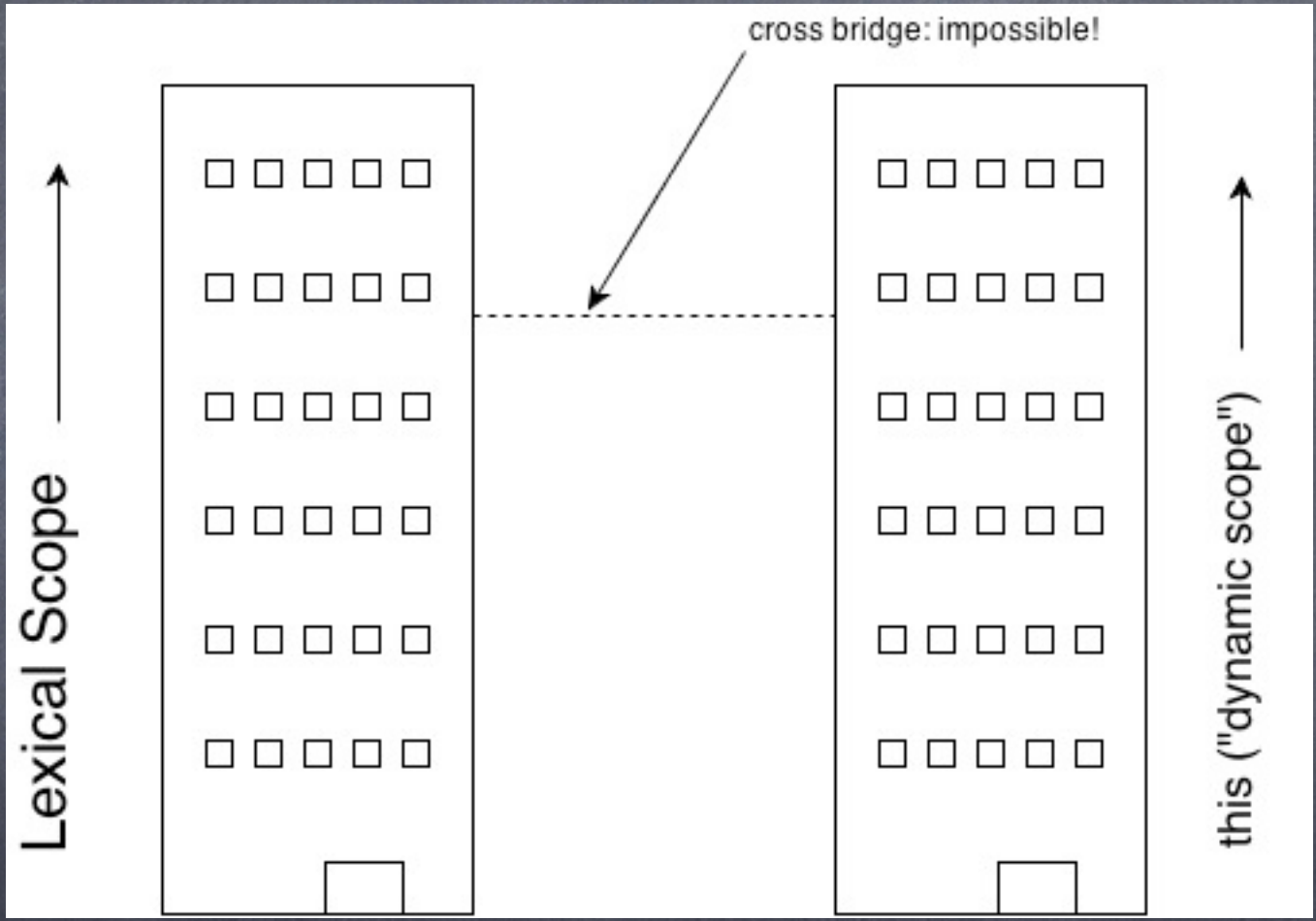
Agenda

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.



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

Code Me

```
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

Code Me

```
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

Code Me

```
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

Code Me

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

Code Me

```
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

Code Me

```
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

Code Me

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?

Information hiding

Principle of Least Exposure

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

Encapsulation

private

~~protected~~

public

Encapsulation

```
var Hello = function(d){
    var name;   // private data!

    // auto-execute "constructor"
    (function Hello(d) {
        name = d.name;
    })(d);

    function speak() {
        console.log("Hello " + name);
    }

    var publicAPI = {
        speak: speak
    };
    return publicAPI;
};

var world = Hello({ name:"world" });
world.speak();  // "Hello world"
```

Encapsulation

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;
```

OO Design Patterns: singleton

```
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);
```

OO Design Patterns: observer

prototype

Every single “object” is built by a constructor call

prototype

A constructor makes an object "~~based on~~" its own prototype

A constructor makes an object linked to its own prototype

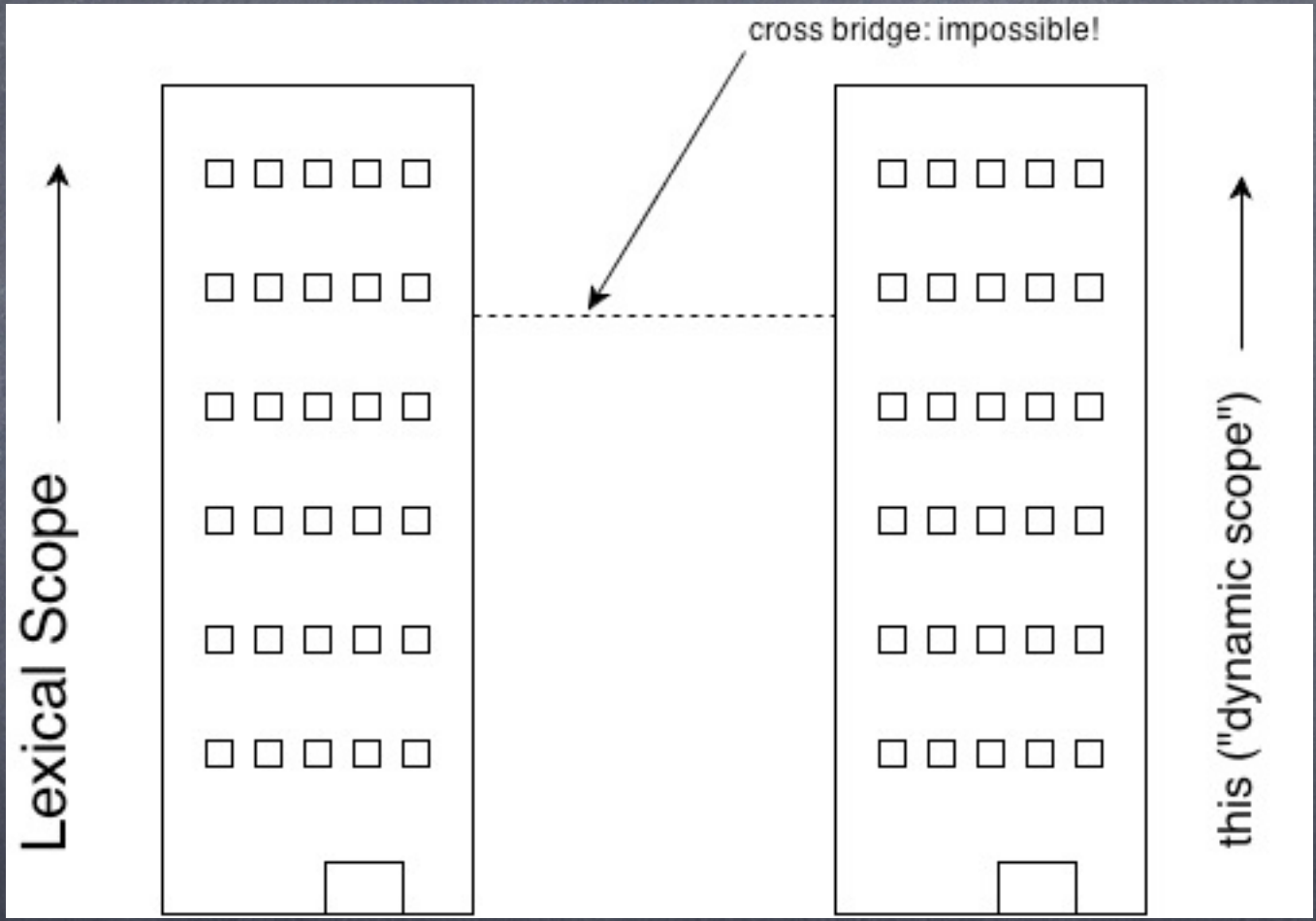
```
1  function Foo(who) {
2      this.me = who;
3  }
4  Foo.prototype.identify = function() {
5      return "I am " + this.me;
6  };
7
8  var a1 = new Foo("a1");
9  var a2 = new Foo("a2");
10
11 a2.speak = function() {
12     alert("Hello, " + this.identify() + ".");
13 };
14
15 a1.constructor === Foo;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

prototype

Code Me

```
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



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."
```

Code Me

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."
```

prototype

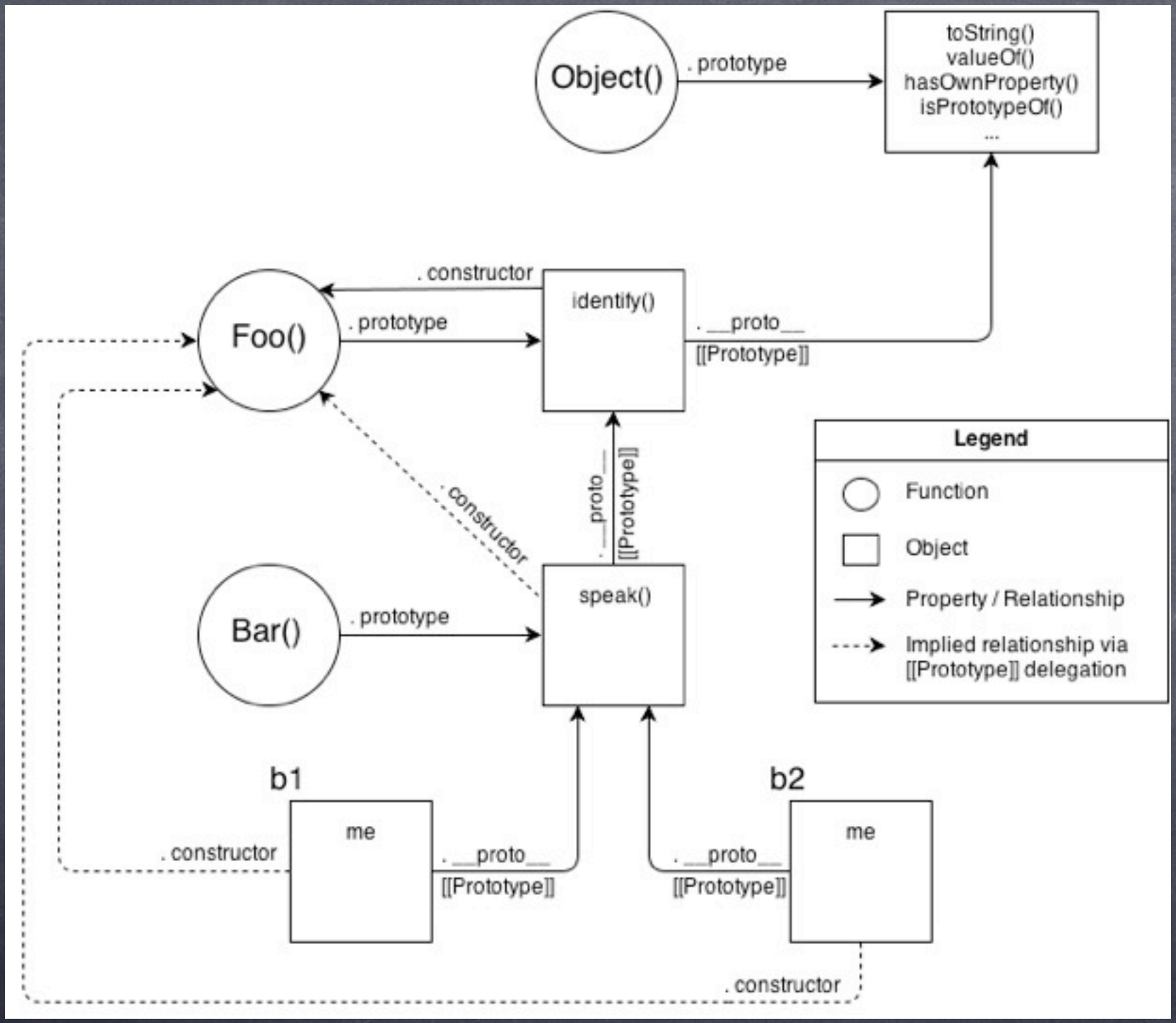
“Inheritance”

prototype

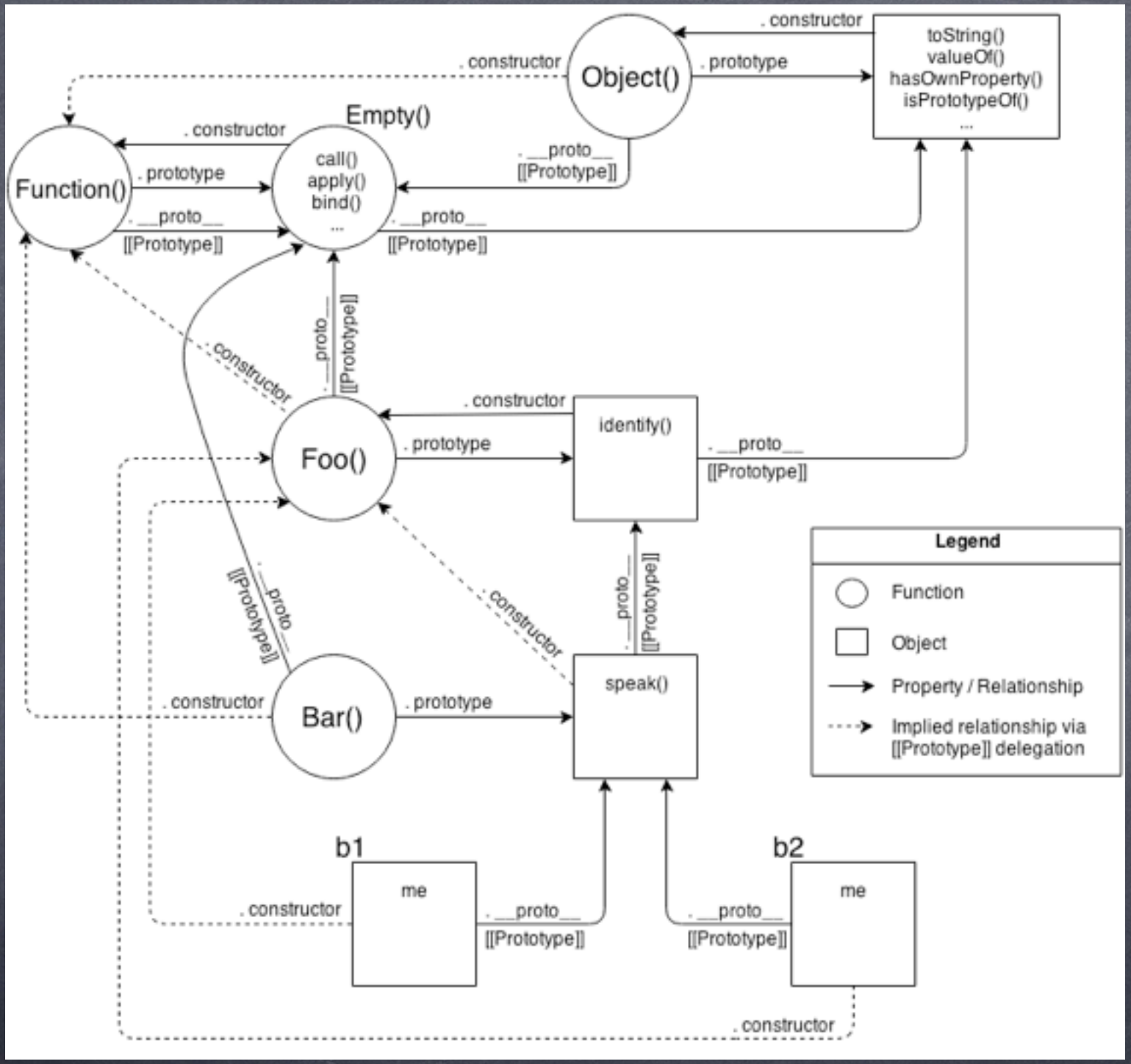
```
1  function Foo(who) {
2      this.me = who;
3  }
4  Foo.prototype.identify = function() {
5      return "I am " + this.me;
6  };
7
8  function Bar(who) {
9      Foo.call(this,who);
10 }
11 // Bar.prototype = new Foo();    // Or...
12 Bar.prototype = Object.create(Foo.prototype);
13 // NOTE: constructor 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

Code Me



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)?

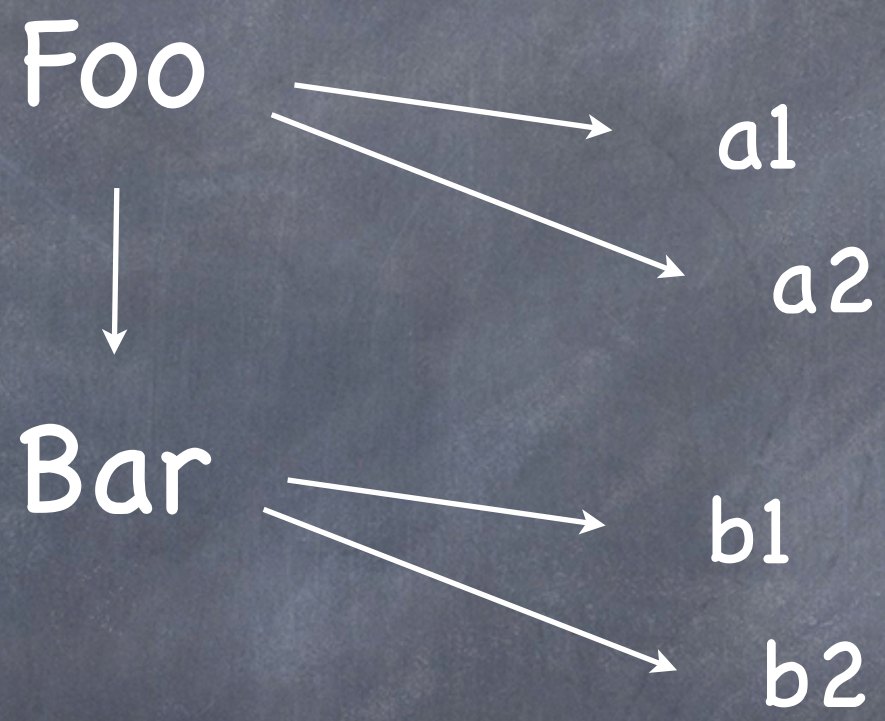
remember how this has a pesky way of getting unassigned?

```
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);
```

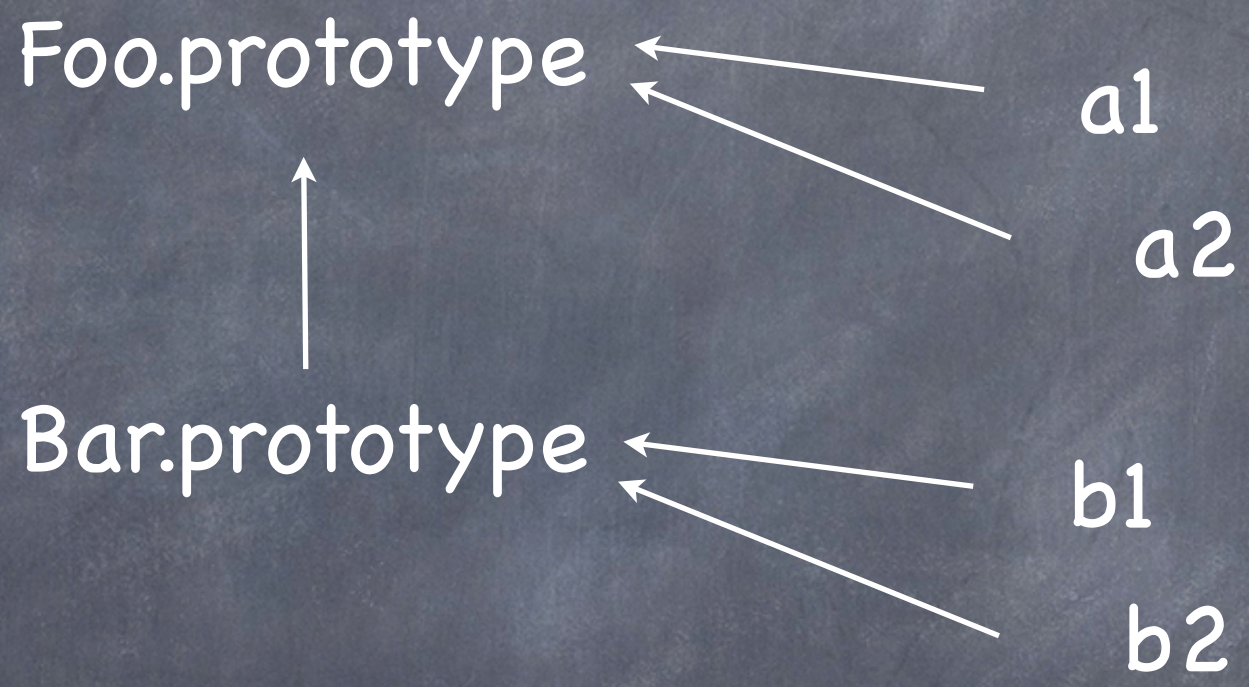
prototype: this redux

(exercise #7: 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!

OLOO:

Objects Linked to Other 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 = new Bar("b1");
18 b1.speak(); // alerts: "Hello, I am b1."
```

OLOO: delegated objects

Code Me

```
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

Code Me

```
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

Code Me

```
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."
```

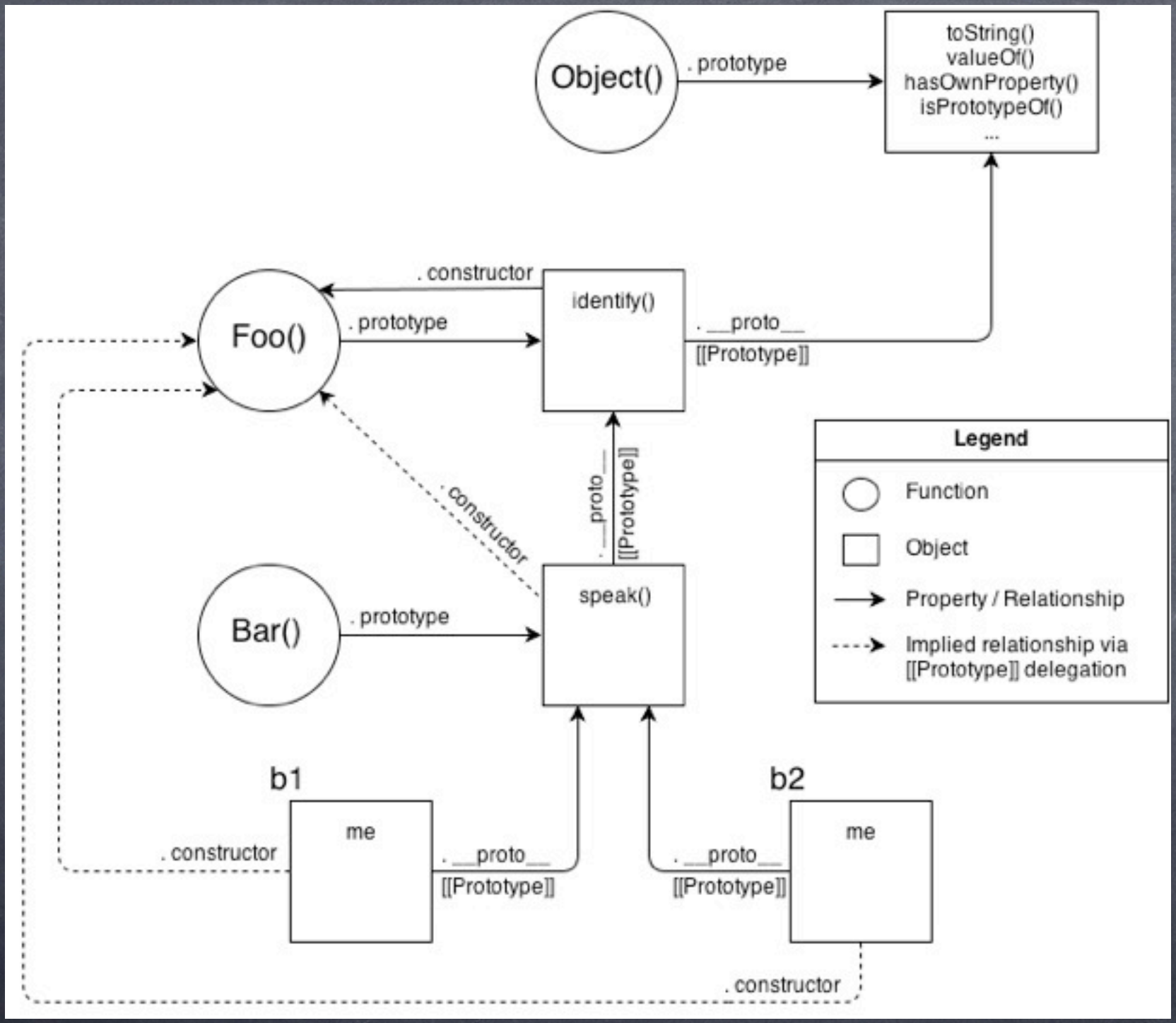
OLOO: delegated objects

Code Me

Mental Models

```
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."
```

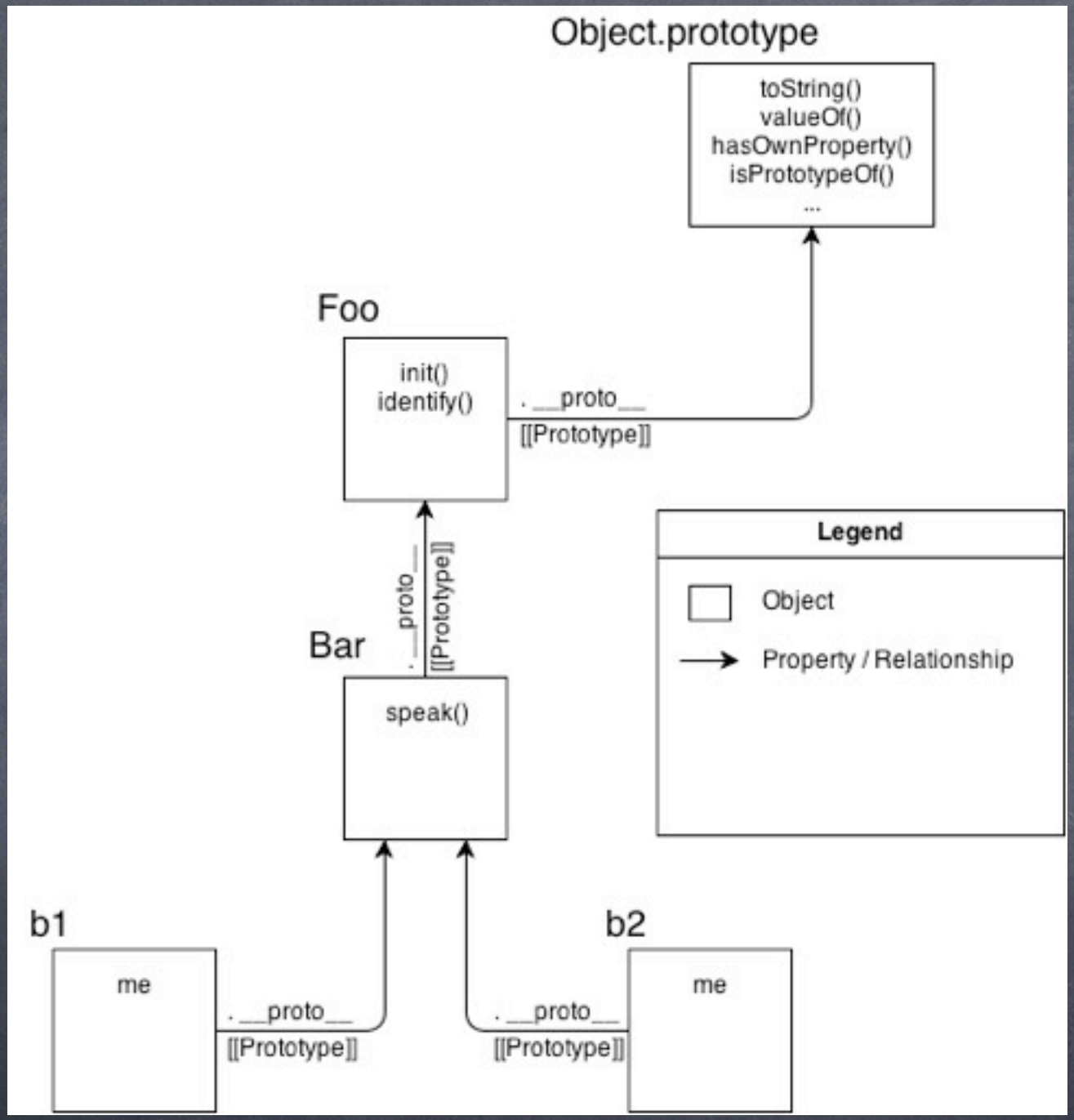
OO: old & busted



OO: old & busted

```
var Foo = {
    init: function(who) {
        this.me = who;
    },
    identify: function() {
        return "I am " + this.me;
    }
};

var Bar = Object.create(Foo);

Bar.speak = function() {
    alert("Hello, " + this.identify() + ".");
};

var b1 = Object.create(Bar);
b1.init("b1");
var b2 = Object.create(Bar);
b2.init("b2");

b1.speak(); // alerts: "Hello, I am b1."
b2.speak(); // alerts: "Hello, I am b2."
```

OLOO: new hotness



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

~~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 #8: 10min)

Kyle Simpson
@getify
http://getify.me

Thanks!

Questions?