KYLE SIMPSON GETIFY@GMAIL.COM

DEEP JS FOUNDATIONS

Scope, Closures

- Nested Scope
- Hoisting
- Closure
- Modules

Scope: where to look for things

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 }
```

Scope

```
1  "use strict";
2
3  var foo = "bar";
4
5  function bar() {
6      var foo = "baz";
7  }
8
9  function baz(foo) {
10     foo = "bam";
11     bam = "yay";     // error!
12 }
```

Scope

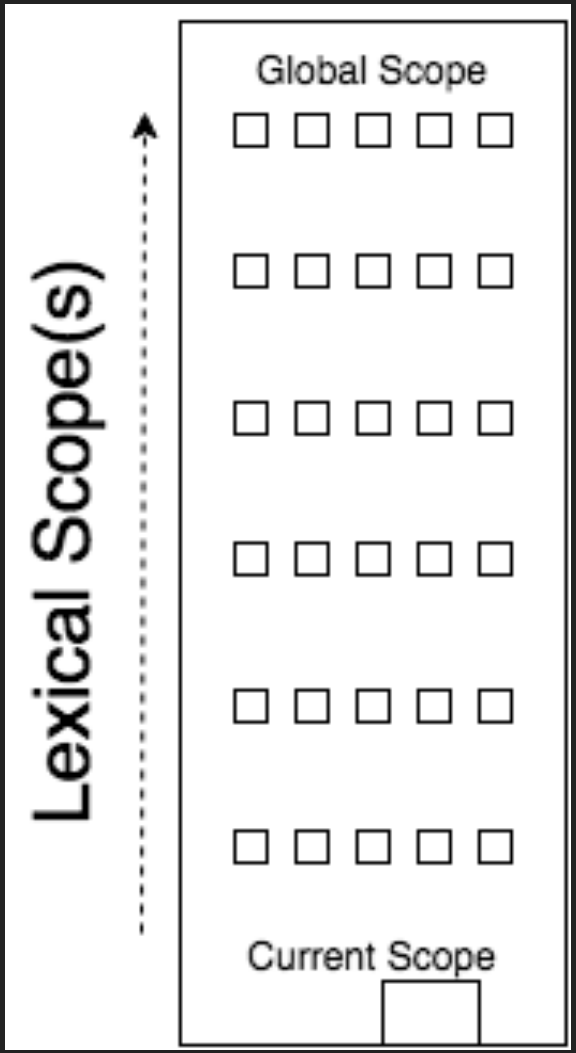
```
1  var foo = "bar";
2
3  function bar() {
4      var foo = "baz";
5
6      function baz(foo) {
7          foo = "bam";
8          bam = "yay";
9      }
10     baz();
11 }
12
13 bar();
14 foo;          // ???
15 bam;          // ???
16 baz();        // ???
```

Scope

Code Me

```
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



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: which scope?

Code Me

```
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: which scope?

Code Me

Named Function Expressions

```
1 var clickHandler = function(){
2     // ..
3 };
4
5 var keyHandler = function keyHandler(){
6     // ..
7 };
```

Named Function Expressions

1. Handy function self-reference

2. More debuggable stack traces

3. More self-documenting code

Named Function Expressions: Benefits

lexical scope

dynamic 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  // 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

Function Scoping

```
1  var foo = "foo";
2
3  // ..
4
5  var foo = "foo2";
6  console.log(foo);    // "foo2"
7
8  // ..
9
10 console.log(foo);    // "foo2" -- oops!
```

Function Scoping

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

Function Scoping

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

Function Scoping

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

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

Function Scoping: IIFE

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

Function Scoping: IIFE

```
1 for (var i = 0; i < 5; i++) {
2     (function IIFE(){
3         var j = i;
4         console.log(j);
5     })();
6 }
```

Function Scoping: IIFE

Block Scoping

```
1 function diff(x,y) {
2     if (x > y) {
3         var tmp = x;
4         x = y;
5         y = tmp;
6     }
7
8     return y - x;
9 }
```

Block Scoping: intent

```
1 function diff(x,y) {
2     if (x > y) {
3         let tmp = x;
4         x = y;
5         y = tmp;
6     }
7
8     return y - x;
9 }
```

Block Scoping: let

```
1  function formatStr(str) {
2      { let prefix, rest;
3          prefix = str.slice( 0, 3 );
4          rest = str.slice( 3 );
5          str = prefix.toUpperCase() + rest;
6      }
7
8      if (/^FOO:/.test( str )) {
9          return str;
10     }
11
12     return str.slice( 4 );
13 }
```

Block Scoping: explicit let block

```
1 function repeat(fn,n) {
2     var result;
3
4     for (var i = 0; i < n; i++) {
5         result = fn( result, i );
6     }
7
8     return result;
9 }
```

Block Scoping: "well, actually, not all vars..."

```
1 function repeat(fn,n) {
2     var result;
3
4     for (let i = 0; i < n; i++) {
5         result = fn( result, i );
6     }
7
8     return result;
9 }
```

Block Scoping: let + var

```
1  function lookupRecord(searchStr) {
2      try {
3          var id = getRecord( searchStr );
4      }
5      catch (err) {
6          var id = -1;
7      }
8
9      return id;
10 }
```

Block Scoping: sometimes var > let

```
1 var a = 2;
2 a++;                    // 3
3
4 const b = 2;
5 b++;                    // Error!
6
7 const c = [2];
8 c[0]++;                 // 3 <--- oops!?
```

Block Scoping: const(antly confusing)

Quiz

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

(exercise #2: 5min)

Hoisting

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

Scope: hoisting

Code Me

```
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

(exercise #3: 5min)

Closure

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

Closure

```
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();
```

Closure

Code Me

```
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();
```

Closure

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

Closure

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

Closure

```
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

Code Me

```
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

Code Me

```
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

Code Me

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

Closure: loops + block scope

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

Closure: loops + block scope

(exercise #4: 5min)

Modules

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

Not a module

```
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"
```

Classic module pattern

Code Me

```
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"
```

Classic module pattern: modified

Code Me

```
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 });
```

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

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 #5: 30min)

Object-Orienting

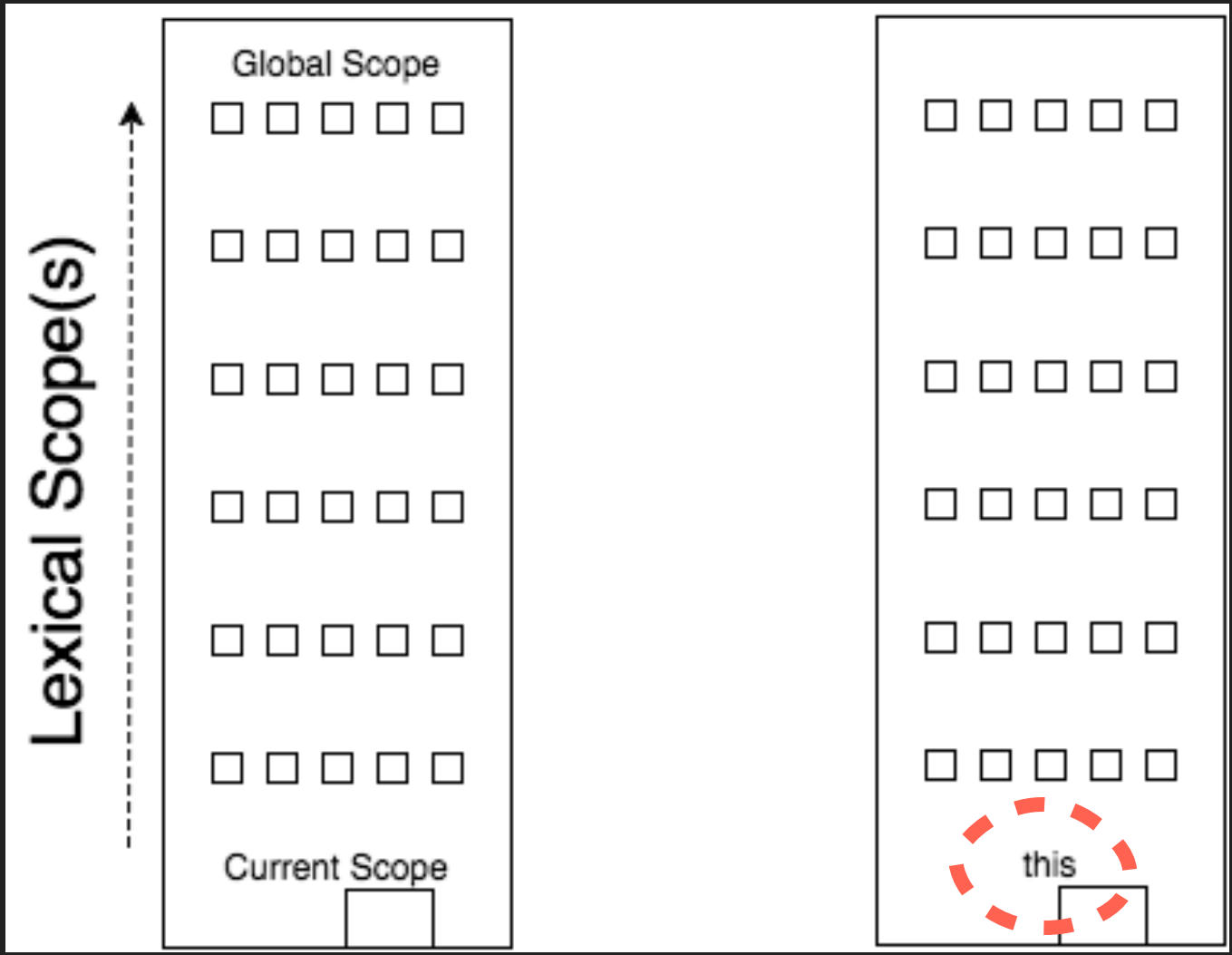
- this
- Prototypes
- class { }
- “Inheritance” vs. “Behavior Delegation”
  (OO vs. OLOO)

this

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

Remember lexical scope vs. dynamic scope?

JavaScript's version of "dynamic scope" is **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 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();    // ???
```

AKA: "constructor call"

this: new binding

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. Is the function called by new?

2. Is the function called by call() or apply()?

   Note: bind() effectively uses apply()

3. Is the function called on a context object?

4. DEFAULT: global object (except strict mode)

this: determination

Quiz

1. How do you “borrow” a function and implicitly set this?
2. How do you explicitly set this for the function call?
3. How can you lock a specific this to a function? Why do that? Why not?
4. How do you create a new this for the function call?

```
1  function foo() {
2      return () => this.bar;
3  }
4
5  var bar = "bar1";
6  var o1 = { bar: "bar2", foo: foo };
7  var o2 = { bar: "bar3" };
8
9  var f1 = foo();
10 var f2 = o1.foo();
11 var f3 = foo.call(o2);
12
13 f1();              // "bar1"
14 f2();              // "bar2"
15 f3();              // "bar3"
16
17 f1.call( o2 );     // "bar1"    <--- hmmmm
```

this: what about => functions?

(exercise #6: 5min)

Prototypes

Objects are built by constructor calls

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

A constructor makes an object linked to its own prototype

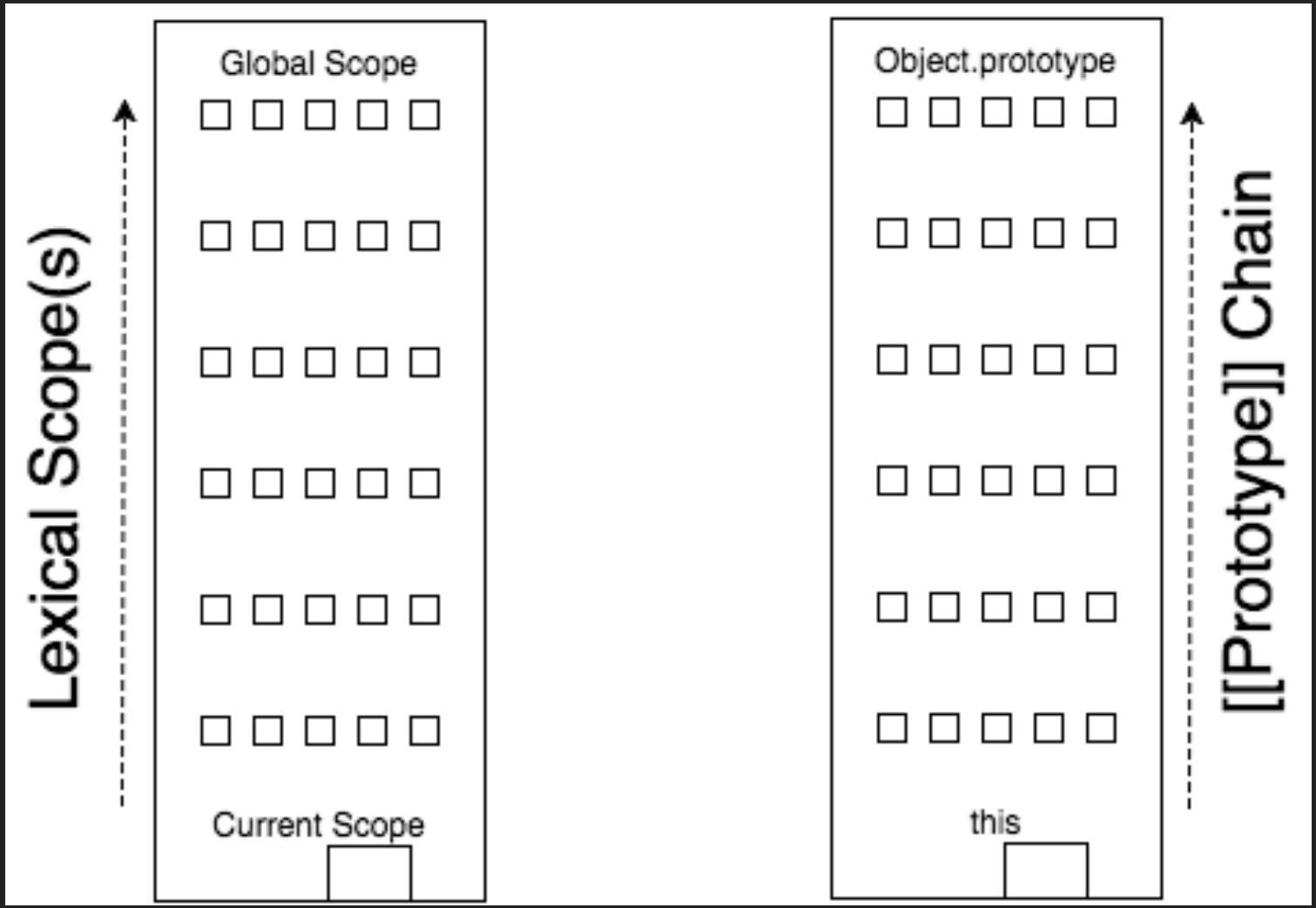
```
1  function Foo(who) {
2      this.me = who;
3  }
4  Foo.prototype.identify = function() {
5      return "I am " + this.me;
6  };
7
8  var a1 = new Foo("a1");
9  var a2 = new Foo("a2");
10
11 a2.speak = function() {
12     alert("Hello, " + this.identify() + ".");
13 };
14
15 a1.constructor === Foo;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

Prototypes

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;
```

Prototypes


Lexical Scope(s)
Global Scope
Current Scope
[[Prototype]] Chain
Object.prototype
this

Prototypes

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

Prototypes: shadowing

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

Prototypes: shadowing

```
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."
```

Prototypes: shadowing

Code Me

```
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."
```

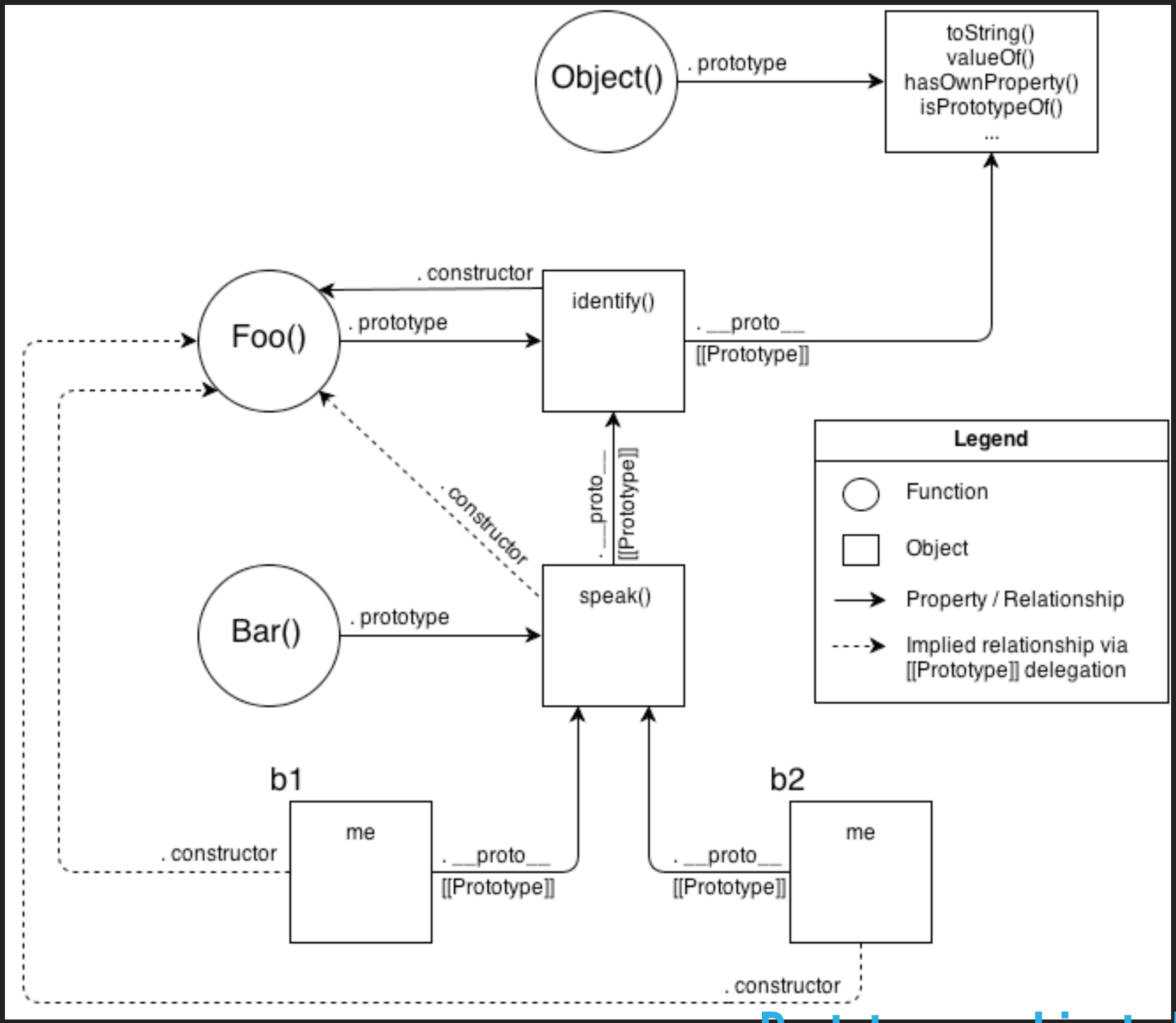
Prototypes: avoid shadowing

“Inheritance”

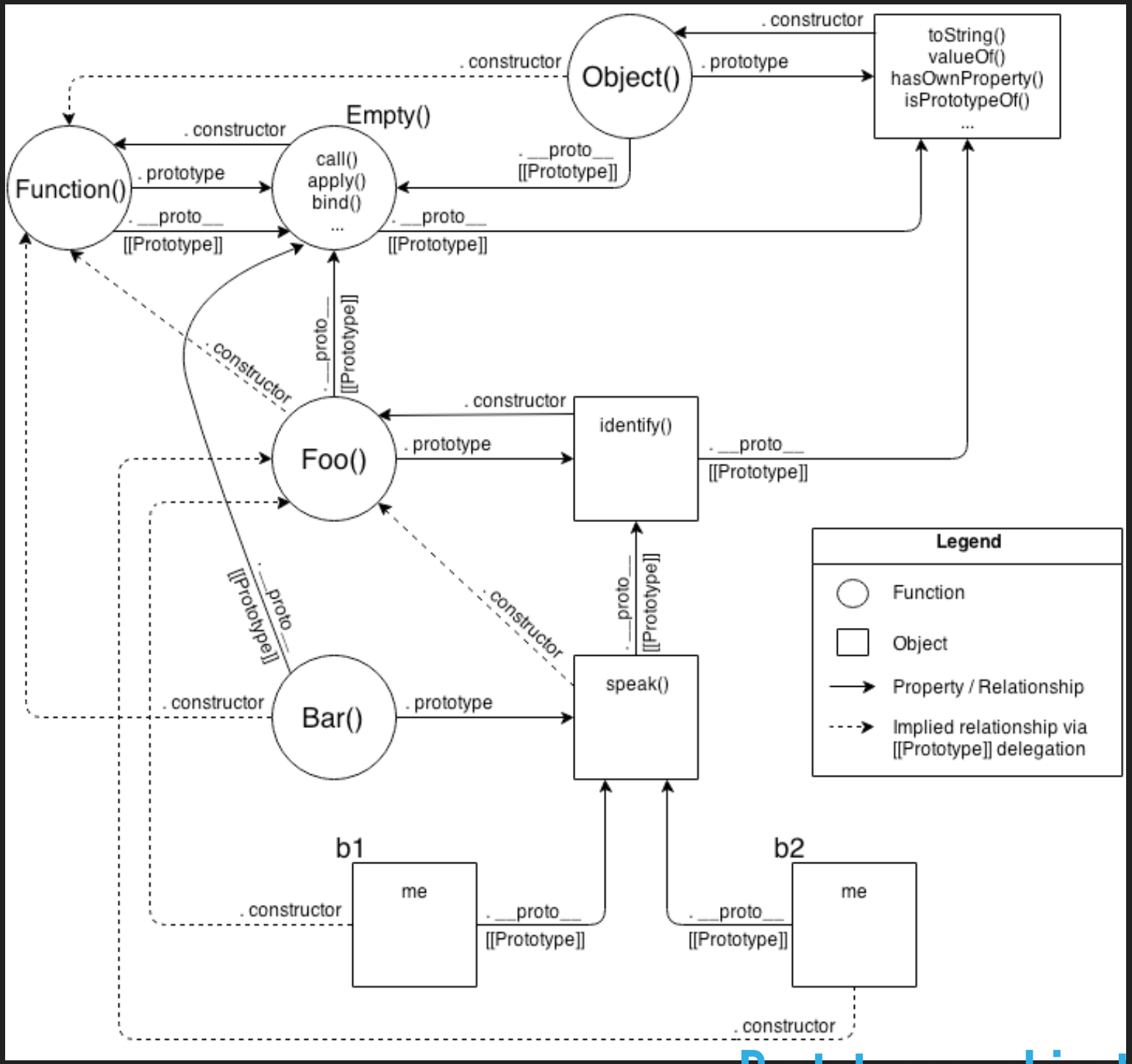
```
1  function Foo(who) {
2      this.me = who;
3  }
4  Foo.prototype.identify = function() {
5      return "I am " + this.me;
6  };
7
8  function Bar(who) {
9      Foo.call(this,who);
10 }
11 // Bar.prototype = new Foo();    // Or...
12 Bar.prototype = Object.create(Foo.prototype);
13 // NOTE: constructor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

Prototypes: objects linked

Code Me



Prototypes: objects linked



Prototypes: 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)?

(exercise #7: 10min)

class { }

ES6

```
1  class Foo {
2      constructor(who) {
3          this.me = who;
4      }
5
6      identify() {
7          return "I am " + this.me;
8      }
9  }
10
11 var a1 = new Foo("a1");
12 var a2 = new Foo("a2");
13
14 a1.identify();  // "I am a1"
15 a2.identify();  // "I am a2"
```

ES6 class

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

ES6 class: extends (inheritance)

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

ES6 class: super (relative polymorphism)

```
1  class Foo {
2      constructor(who) {
3          this.me = who;
4      }
5
6      identify() {
7          return "I am " + this.me;
8      }
9
10     static hello() { return "Hello!"; }
11 }
12
13 class Bar extends Foo {
14     speak() {
15         alert("Hello, " + this.identify() + ".");
16     }
17 }
18
19 Foo.hello();     // Hello!
20
21 Bar.hello();     // Hello!
```

ES6 class: static (constructor inheritance)

```
class Caution { }
```

ES6 class

```
1 var Foo = {
2     // ..
3 };
4
5 class Bar extends Foo { // <--- error!
6     // ..
7 }
```

ES6 class: caution!

```
1  class Foo {
2      constructor() {
3          alert("Hello!");
4      }
5  }
6
7  class Bar {
8      constructor(who) {
9          this.me = who;
10     }
11 }
12
13 Foo();            // <--- error!
14
15 Bar.call({});     // <--- error!
```

ES6 class: caution!

```
1   class Foo {
2       constructor(who) {
3           this.me = who;
4       }
5
6       identify() {
7           return "I am " + this.me;
8       }
9   }
10
11  class Bar extends Foo {
12      constructor(who) {
13          this.x = 1;          // <--- error!
14          super( who );        // <--- this must come first!
15      }
16  }
17
18  var b1 = new Bar("b1");
```

ES6 class: caution!

```
1   class A {
2       one() { console.log("one:A"); }
3       two() { console.log("two:A"); }
4   }
5
6   var B = {
7       one() { console.log("one:B"); },
8       two() { console.log("two:B"); }
9   };
10
11  class C extends A {
12      foo() {
13          this.one();
14          super.two();
15      }
16  }
17
18  var x = new C();
19
20  x.foo();                // one:A two:A
21
22  x.foo.call( B );        // one:B two:A  <--- Oops!
```

ES6 class: caution!

```
1  class A {
2      one() { console.log("one:A"); }
3      two() { console.log("two:A"); }
4  }
5
6  class B extends A { }
7
8  B.prototype.foo = function() {
9      this.one();
10     super.two();          // syntax error!
11 };
```

ES6 class: caution!

```
1  class A {
2      one() { console.log("one:A"); }
3      two() { console.log("two:A"); }
4  }
5
6  var B = {
7      __proto__: A.prototype,
8      foo() {
9          this.one();
10         super.two();
11     },
12     bar: function() {
13         this.one();
14         super.two();          // syntax error!
15     }
16 };
```

ES6 class: caution!

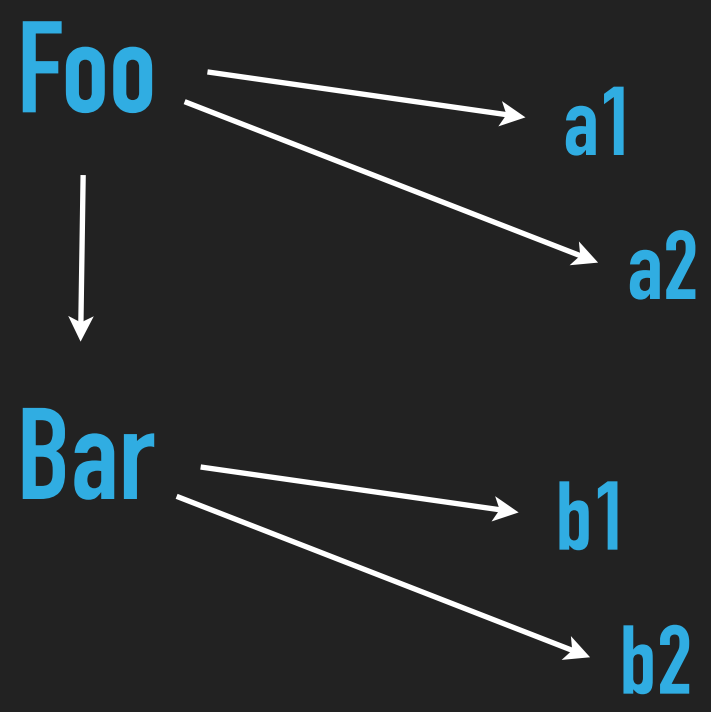
```
1  class A {
2      one() { console.log("one:A"); }
3      two() { console.log("two:A"); }
4  }
5
6  var B = Object.assign(Object.create(A.prototype),{
7      foo() {
8          this.one();
9          super.two();          // error, two doesn't exist!
10     }
11 });
12
13 B.foo();
```

ES6 class: caution!

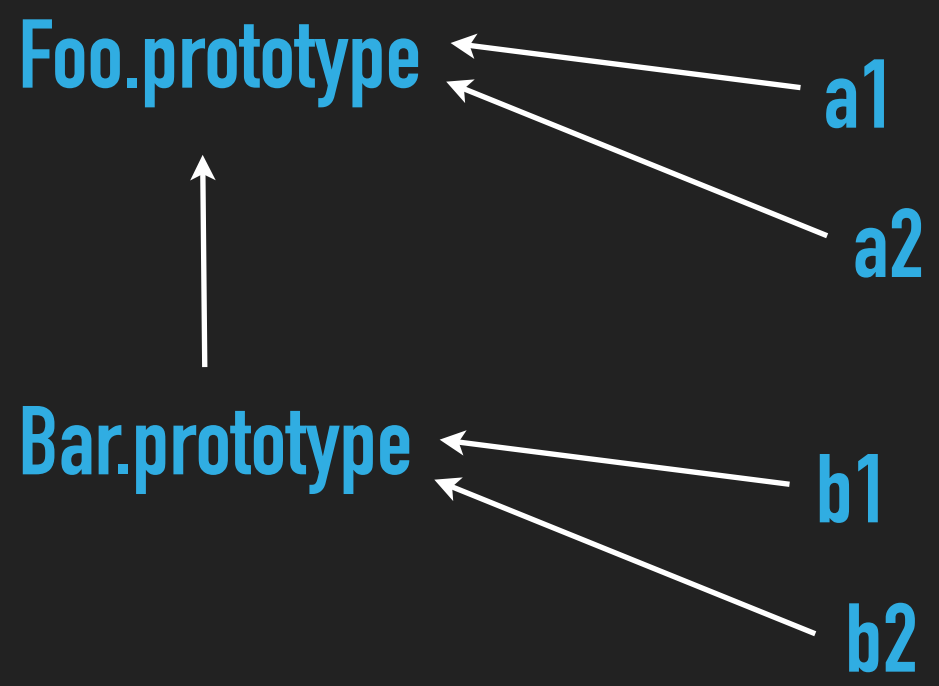
```
class NotSoSuper { }
```

ES6 class: super frustrating!

Clearing Up Inheritance



OO: classical inheritance



(another design pattern)

OO: “prototypal inheritance”

JavaScript ~~"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

```
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: delegated objects

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

OLOO: Object.create()

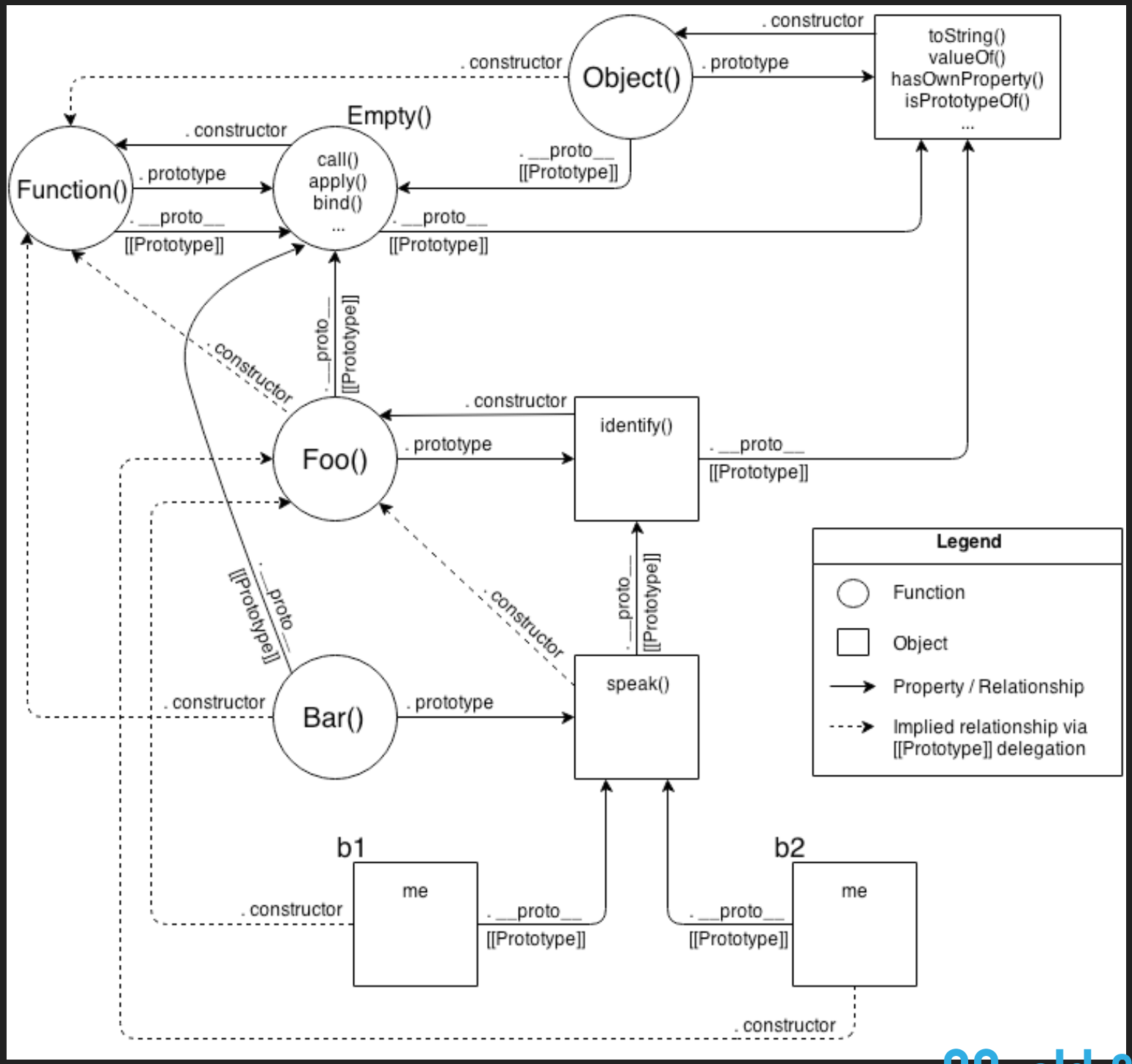
```
1 if (!Object.create) {
2     Object.create = function (o) {
3         function F() {}
4         F.prototype = o;
5         return new F();
6     };
7 }
```

OLOO: Object.create()

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



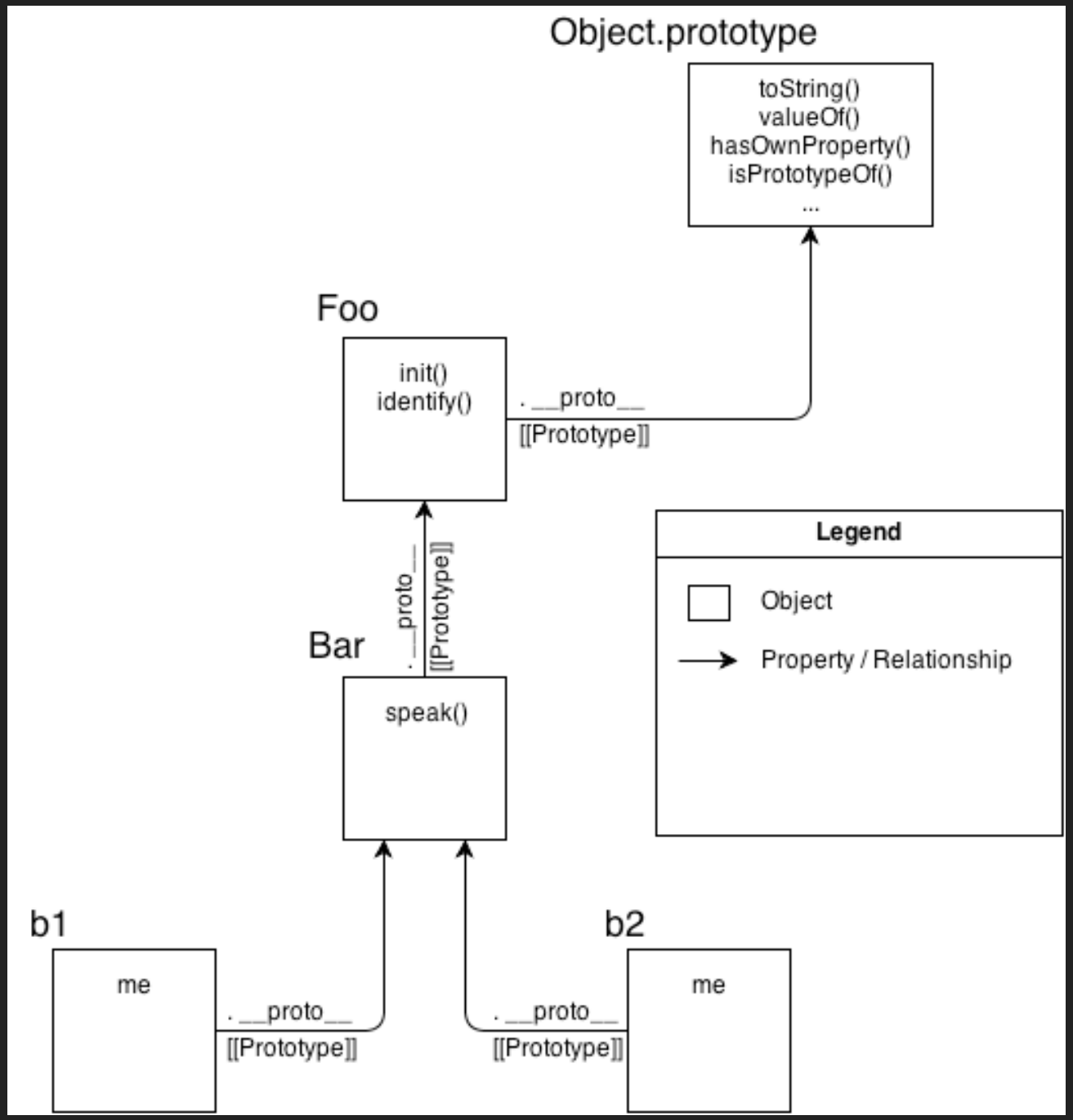
OO: old & busted

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

ES6 class: better?

```
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



OLOO: new hotness

Delegation: Design Pattern

~~Parent-Child~~

Peer-Peer

Delegation-Oriented Design

```
1  var AuthController = {
2      authenticate() {
3          server.authenticate(
4              [ this.username, this.password ],
5              this.handleResponse.bind(this)
6          );
7      },
8      handleResponse(resp) {
9          if (!resp.ok) this.displayError(resp.msg);
10     }
11 };
12
13 var LoginFormController =
14     Object.assign(Object.create(AuthController),{
15         onSubmit() {
16             this.username = this.$username.val();
17             this.password = this.$password.val();
18             this.authenticate();
19         },
20         displayError(msg) {
21             alert(msg);
22         }
23     });
```

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]]?

Types, Coercion

- Primitive Types
- Natives
- Coercion
- Equality

- undefined
- string
- number
- boolean
- object
- function
- null

Primitive Types

```
1 typeof foo;            // "undefined"
2 typeof "foo";          // "string"
3 typeof 123;            // "number"
4 typeof true;           // "boolean"
5 typeof { a: 1 };       // "object"
6 typeof function() { alert(e); };    // "function"
```

Primitive Types (literals)

```
2
3 typeof null;    // "object" ... WAT!?
4
```

Primitive Types (literals)

Quiz

```
1 var foo;
2 typeof foo;                    // ???
3
4 var bar = typeof bar;
5 bar;                           // ???
6 typeof bar;                    // ???
7
8 typeof typeof 2;               // ???
```

Quiz

```
1 var foo;
2 typeof foo;                    // "undefined"
3
4 var bar = typeof bar;
5 bar;                           // "undefined"
6 typeof bar;                    // "string"
7
8 typeof typeof 2;               // "string"
```

Primitive Types

- NaN ("not a number")
- Infinity, -Infinity
- null
- undefined (void)
- +0, -0

Primitive Types: special values

NaN

```
1 var a = "a" / 2;
2
3 a;                    // NaN
4 typeof a;             // "number"
5 isNaN(a);             // true
6
7 isNaN("foo");         // true! WAT!?
```

NaN

```
1 if (!Number.isNaN) {
2     Number.isNaN = function isNaN(num){
3         return (
4             typeof num == "number" &&
5             window.isNaN(num)
6         );
7     };
8 }
```

Primitive Types: special values

NaN

```
1 if (!Number.isNaN) {
2     Number.isNaN = function isNaN(num){
3         return num != num;
4     };
5 }
```

Primitive Types: special values

Negative Zero

```
1 var foo = 0 / -3;
2 foo === -0;              // true ... yay?
3 foo === 0;               // true ... doh!
4 0 === -0;                // true ... grr!
5 (0/-3) === (0/3);        // true ... sigh
6
7 foo;                     // 0 ... WAT!?
```

Negative Zero

```
1 function isNeg0(x) {
2   return x===0 && (1/x)===-Infinity;
3 }
4
5 isNeg0(0 / -3);      // true ... yay!
6 isNeg0(0 / 3);       // false ... yay!
```

ES6: Object.is(..)

```
1 Object.is( "foo", NaN );      // false
2 Object.is( NaN, NaN );        // true
3
4 Object.is( 0, -0 );           // false
5 Object.is( -0, -0 );          // true
```

Primitive Types: special values

Quiz

```
1  var baz = 2;
2  typeof baz;                    // ???
3  var baz;
4  typeof baz;                    // ???
5  baz = null;
6  typeof baz;                    // ???
7
8  baz = "baz" * 3;
9  baz;                           // ???
10 typeof baz;                    // ???
11
12 baz = 1 / 0;
13 baz;                           // ???
14 typeof baz;                    // ???
```

Primitive Types: special values

Quiz

```
1  var baz = 2;
2  typeof baz;                    // "number"
3  var baz;
4  typeof baz;                    // "number"
5  baz = null;
6  typeof baz;                    // "object"
7
8  baz = "baz" * 3;
9  baz;                           // NaN
10 typeof baz;                    // "number"
11
12 baz = 1 / 0;
13 baz;                           // Infinity
14 typeof baz;                    // "number"
```

Primitive Types: special values

- String
- Number
- Boolean
- Function
- Object
- Array
- RegExp
- Date
- Error

Native Types?
Native Objects?

Native Constructors?

- new String()
- new Number()
- new Boolean()
- new Function()
- new Object()
- new Array()
- new RegExp()
- new Date()
- new Error()

Native Functions

- String()
- Number()
- Boolean()
- Function()
- Object()
- Array()
- RegExp()
- Date()
- Error()

Quiz

```
1  var foo = new String("foo");
2  foo;                                  // ???
3  typeof foo;                           // ???
4  foo instanceof String;                // ???
5  foo instanceof string;                // ???
6
7  foo = String("foo");
8  typeof foo;                           // ???
9
10 foo = new Number(37);
11 typeof foo;                           // ???
```

Quiz

```
1  var foo = new String("foo");
2  foo;                                  // ???
3  typeof foo;                           // "object"
4  foo instanceof String;                // true
5  foo instanceof string;                // false; ref error!
6
7  foo = String("foo");
8  typeof foo;                           // "string"
9
10 foo = new Number(37);
11 typeof foo;                           // "object"
```

Natives

```
1  var foo;
2
3  foo = new Array(1,2,3); // don't!
4
5  foo = [1,2,3];          // do!
6
7  foo = new Object();     // don't!
8  foo.a = 1;
9  foo.b = 2;
10 foo.c = 3;
11
12 foo = { a:1, b:2, c:3 };// do!
```

Natives: literals

```
1 var foo;
2
3 foo = new RegExp("a*b","g");
4
5 foo = /a*b/g;
6
7 foo = new Date();
8
9 // no date literal!
```

Natives: literals

Coercion

Abstract Operations

ToString

null	"null"
undefined	"undefined"
true	"true"
false	"false"
3.14159	"3.14159"
0	"0"
-0	"0"

Coercion: ToString

ToString: toString()

[]	""
[1,2,3]	"1,2,3"
[null,undefined]	",,"
[[[],[],[]],[]]	",,,,"
[,,,,]	",,,,"

Coercion: ToString (Array)

{}	"[object Object]"
{a:2}	"[object Object]"

Coercion: ToString (Object)

ToNumber

""	0
"0"	0
"-0"	-0
" 009 "	9
"3.14159"	3.14159
"0."	0
".0"	0
"."	NaN
"0xaf"	175

Coercion: ToNumber

false	0
true	1
null	0
undefined	NaN

Coercion: ToNumber

ToNumber (object):
ToPrimitive (number)

Coercion: ToNumber (Array/Object)

ToPrimitive

valueOf()
toString()

Coercion: ToPrimitive

[""]	0
["0"]	0
["-0"]	-0
[null]	0
[undefined]	0
[1,2,3]	NaN
[[[[]]]]	0

Coercion: ToNumber/ToPrimitive (Array)

ToBoolean

Falsy	Truthy
""	"foo"
0, +0, -0	23
null	{ a:1 }
NaN	[1,3]
false	true
undefined	function(){..}
	...

Coercion: ToBoolean

Explicit

Implicit

Coercion

Explicit: it's obvious from the code that you're doing it

Explicit Coercion

string ⟷ number

Explicit Coercion

```
1  var foo = "123";
2  var baz = parseInt(foo,10);
3  baz;                          // 123
4
5  baz = Number(foo);
6  baz;                          // 123
7
8  baz = +foo;                   // explicit?
9  baz;                          // 123
10
11 baz = 456;
12 foo = baz.toString();
13 foo;                          // "456"
14
15 foo = String(baz);
16 foo;                          // "456"
```

Explicit Coercion

$$* \longrightarrow \text{boolean}$$

Explicit Coercion

```
1  var foo = "123";
2  var baz = Boolean(foo);
3  baz;                                  // true
4
5  baz = !!foo;
6  baz;                                  // true
7
8  // explicitly implicit!
9  baz = foo ? true : false;
10 baz;                                  // true
```

Explicit Coercion

```
1 var now = +new Date();
2 // now = Date.now(); // ES5 only!
3
4 var foo = "foo";
5 // ~N -> -(N+1)
6 if (~foo.indexOf("f")) {
7     alert("Found it!");
8 }
```

Explicit (or Implicit?) Coercion

http://davidwalsh.name/fixing-coercion

Number("") === 0
Number(false) === 0
Number(true) === 1
Number(null) === 0

String([]) === ""
String([null]) === ""
String([undefined]) === ""

Value Coercion: The Bad Parts

http://getify.github.io/coercions-grid/

	String(x)	x + ''	x.toString()	Number(x), +x	x * 1	x + 0
0	'0'	'0'	'0'	0	0	0
.0	'0'	'0'	'0'	0	0	0
-0	'0'	'0'	'0'	-0	-0	0
NaN	'NaN'	'NaN'	'NaN'	NaN	NaN	NaN
''	''	''	''	0	0	'0'
' '	' '	' '	' '	0	0	' 0'
'\n\n'	'\n\n'	'\n\n'	'\n\n'	0	0	'\n\n0'
null	'null'	'null'		0	0	0
undefined	'undefined'	'undefined'		NaN	NaN	NaN
true	'true'	'true'	'true'	1	1	1
false	'false'	'false'	'false'	0	0	0
/ /	'/ /'	'/ /'	'/ /'	NaN	NaN	'/ /0'
Infinity	'Infinity'	'Infinity'	'Infinity'	Infinity	Infinity	Infinity
-Infinity	'-Infinity'	'-Infinity'	'-Infinity'	-Infinity	-Infinity	-Infinity
'Infinity'	'Infinity'	'Infin Infinity + ''	'Infinity'	Infinity	Infinity	'Infinity0'
'-Infinity'	'-Infinity'	'-Infinity'	'-Infinity'	-Infinity	-Infinity	'-Infinity0'
function(){}	'function (){}'	'function (){}'	'function (){}'	NaN	NaN	'function (){}0'
Symbol('')	'Symbol()'		'Symbol()'			
[]	''	''	''	0	0	'0'
[0]	'0'	'0'	'0'	0	0	'00'
[.0]	'0'	'0'	'0'	0	0	'00'
[-0]	'0'	'0'	'0'	0	0	'00'
[NaN]	'NaN'	'NaN'	'NaN'	NaN	NaN	'NaN0'
[''']	''	''	''	0	0	'0'
[' ']	' '	' '	' '	0	0	' 0'
['\n\n']	'\n\n'	'\n\n'	'\n\n'	0	0	'\n\n0'
[null]	''	''	''	0	0	'0'

Value Coercion: The Bad Parts

Implicit: happens as a side effect of some other operation

Implicit Coercion

"Any sufficiently advanced technology is indistinguishable from magic."

--Arthur C. Clarke

Implicit Coercion

"Any sufficiently complex technology is indistinguishable from magic."

--many devs

Implicit Coercion

"Any sufficiently **confusing** technology is indistinguishable from magic."

--many devs

Implicit Coercion

Implicit: evil?

```
1  var foo = "123";
2  var baz = foo - 0;
3  baz;                    // 123
4
5  baz = foo - "0";
6  baz;                    // 123
7
8  baz = foo / 1;
9  baz;                    // 123
10
11 baz = 456;
12 foo = baz + "";
13 foo;                    // "456"
14
15 foo = baz - "";
16 foo;                    // 456 ... WAT!?
```

Implicit Coercion

```
1  var foo = "123";
2  if (foo) {                          // yup
3      alert("Sure.");
4  }
5
6  foo = 0;
7  if (foo) {                          // nope
8      alert("Right.");
9  }
10 if (foo == false) {                 // yup
11     alert("Yeah.");
12 }
13
14 var baz = foo || "foo";
15 baz;                                // "foo"
```

Implicit Coercion

```
1  var foo = "123";
2  if (foo == true) {                    // nope!
3      alert("WAT!?");
4  }
5
6  foo = [];
7  if (foo) {                            // yup
8      alert("Sure.");
9  }
10 if (foo == false) {                   // yup!
11     alert("WAT!?");
12 }
```

Implicit Coercion

== coercive equality

```
"0" == null;            // false
"0" == undefined;       // false
"0" == false;           // true -- UH OH!
"0" == NaN;             // false
"0" == 0;               // true
"0" == "";              // false

false == null;          // false
false == undefined;     // false
false == NaN;           // false
false == 0;             // true -- UH OH!
false == "";            // true -- UH OH!
false == [];            // true -- UH OH!
false == {};            // false

"" == null;             // false
"" == undefined;        // false
"" == NaN;              // false
"" == 0;                // true -- UH OH!
"" == [];               // true -- UH OH!
"" == {};               // false

0 == null;              // false
0 == undefined;         // false
0 == NaN;               // false
0 == [];                // true -- UH OH!
0 == {};                // false
```

7/24

17/24

Implicit Coercion

```
1 "0" == false;        // true -- UH OH!
2 false == 0;          // true -- UH OH!
3 false == "";         // true -- UH OH!
4 false == [];         // true -- UH OH!
5 "" == 0;             // true -- UH OH!
6 "" == [];            // true -- UH OH!
7 0 == [];             // true -- UH OH!
```

NEVER use == false or == true!

```
1 [] == ![];        // true
2
```

Ask Yourself:

1. Can either value be true or false?
2. Can either value ever be [], “”, or 0?

```
42 == "43";                    // false
"foo" == 42;                   // false
"true" == true;                // false

42 == "42";                    // true
"foo" == [ "foo" ];            // true
```

Implicit Coercion: The Safe Parts

Primitive ⟷ Native

Implicit Coercion

```
1 var foo = "123";
2 foo.length;                    // 3
3
4 foo.charAt(2);                 // "3"
5
6 foo = new String("123");
7 var baz = foo + "";
8 typeof baz;                    // "string"
```

Implicit Coercion

Coercive Equality

== vs. ===

Implicit Coercion

~~== checks value~~

~~=== checks value and type~~

== allows coercion

=== disallows coercion

Coercive Equality

7.2.13 Abstract Equality Comparison

The comparison x == y, where x and y are values, produces **true** or **false**. Such a comparison is performed as follows:

1. If Type(x) is the same as Type(y), then
   a. Return the result of performing Strict Equality Comparison x === y.
2. If x is **null** and y is **undefined**, return **true**.
3. If x is **undefined** and y is **null**, return **true**.
4. If Type(x) is Number and Type(y) is String, return the result of the comparison x == ToNumber(y).
5. If Type(x) is String and Type(y) is Number, return the result of the comparison ToNumber(x) == y.
6. If Type(x) is Boolean, return the result of the comparison ToNumber(x) == y.
7. If Type(y) is Boolean, return the result of the comparison x == ToNumber(y).
8. If Type(x) is either String, Number, or Symbol and Type(y) is Object, return the result of the comparison x == ToPrimitive(y).
9. If Type(x) is Object and Type(y) is either String, Number, or Symbol, return the result of the comparison ToPrimitive(x) == y.
10. Return **false**.

Coercive Equality: == vs. ===

```
1  var foo = [];
2  var baz = "";
3  if (foo == baz) {          // yup!
4      alert("Doh!");
5  }
6  if (foo === baz) {         // nope
7      alert("Phew.");
8  }
9
10 foo = 0;
11 if (foo == "") {           // yup!
12     alert("Argh!");
13 }
14 if (foo === "") {          // nope
15     alert("Phew.");
16 }
```

Coercive Equality: == vs. ===

but... but...

```
1  var foo = "3";
2  if (foo === 3 || foo === "3") { // yup
3      alert("Thanks, but...");
4  }
5
6  if (foo == 3) {                 // yup!
7      alert("That's nicer!");
8  }
9
10 if (typeof foo === "string") {  // yup
11     alert("typeof always returns a string");
12 }
```

Coercive Equality: == more readable?

```
1  var foo;
2  if (foo == null) {        // yup!
3      alert("Thanks!");
4  }
5
6  foo = null;
7  if (foo == null) {        // yup
8      alert("Thanks, again!");
9  }
10
11 foo = false;
12 if (foo == null) {        // nope!
13     alert("Phew!");
14 }
```

Coercive Equality: == more readable?

What about the performance of == or ===?

7.2.13 Abstract Equality Comparison

The comparison x == y, where x and y are values, produces **true** or **false**. Such a comparison is performed as follows:

1. If Type(x) is the same as Type(y), then
   a. Return the result of performing Strict Equality Comparison x === y.
2. If x is **null** and y is **undefined**, return **true**.
3. If x is **undefined** and y is **null**, return **true**.
4. If Type(x) is Number and Type(y) is String, return the result of the comparison x == ToNumber(y).
5. If Type(x) is String and Type(y) is Number, return the result of the comparison ToNumber(x) == y.
6. If Type(x) is Boolean, return the result of the comparison ToNumber(x) == y.
7. If Type(y) is Boolean, return the result of the comparison x == ToNumber(y).
8. If Type(x) is either String, Number, or Symbol and Type(y) is Object, return the result of the comparison x == ToPrimitive(y).
9. If Type(x) is Object and Type(y) is either String, Number, or Symbol, return the result of the comparison ToPrimitive(x) == y.
10. Return **false**.

Implicit Coercion: == vs. === performance

```
1 var x = 2;
2 var y = 2;
3 var z = "2";
4
5 x == y;
6 x == z;      // slower
7
8 x === y;
9 x === z;     // same
```

Implicit Coercion: == vs. === performance

```
1 // not equivalent:
2 x == y;
3 x === y;
4
5 // equivalent:
6 (x === y || x === z);
```

Implicit Coercion: == vs. === performance

7

- If the types compared are the same, ***they are identical***. That is to say they use ***the exact same algorithm***.
- If the types are *different*, then performance is irrelevant. Either you need type coercion, or you don't. If you don't need it, don't use `==` because the result you get may be unexpected.

share | edit | flag

answered **Nov 8 '11 at 1:18**

user1034768

add comment

Implicit Coercion: == vs. === performance

#FUD: using == in your code is dangerous. Avoid it!

#protip: use === where it's safer and use == where it's more helpful.

#FUD: JavaScript's implicit coercion is a flaw in the design of the language. Avoid it!

#protip: use explicit coercion where it's safer and use implicit coercion where it's more helpful.

(exercise #1: 15min)

(exercise #8: 20min)

Know Your JavaScript

THANKS!!!!

KYLE SIMPSON GETIFY@GMAIL.COM

DEEP JS FOUNDATIONS