



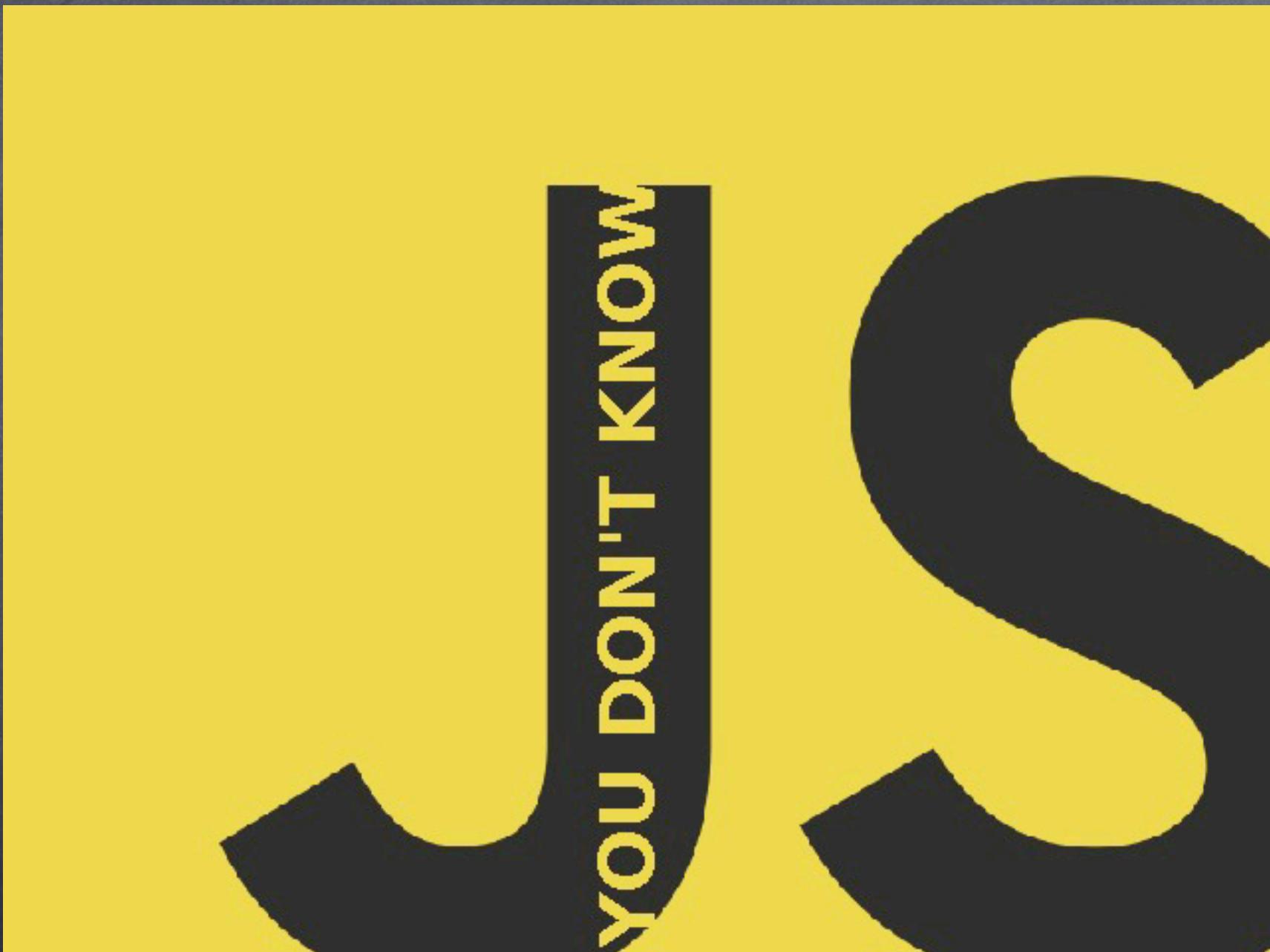
Kyle Simpson  
@getify  
<http://getify.me>

- LABjs
- grips
- asynquence

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

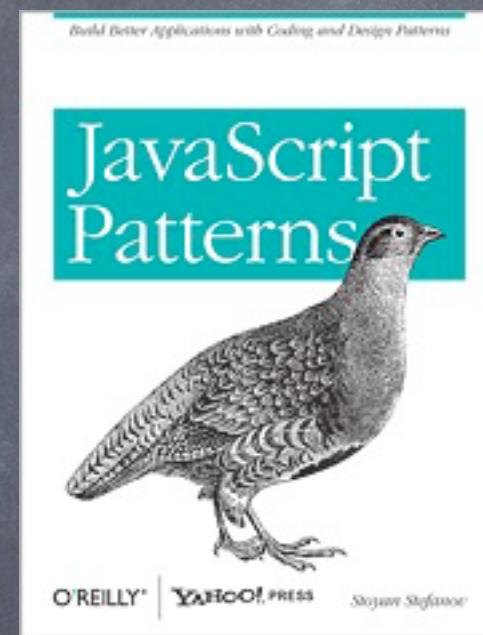
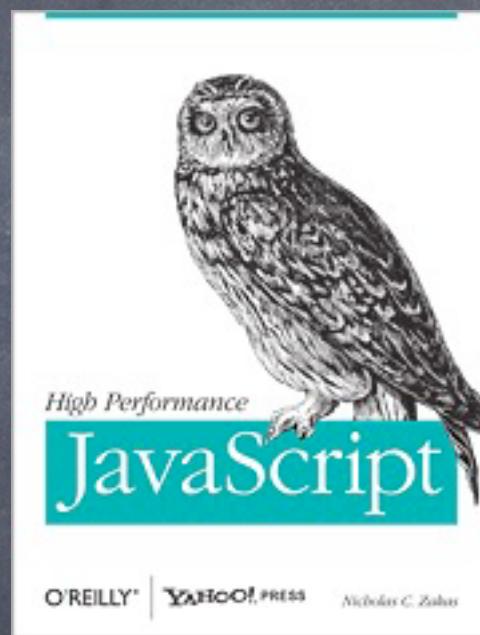


<http://speakerdeck.com/getify>



<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

## JavaScript Background

- Brendan Eich, Netscape
- ES3
- ES5
- ES6+

Brendan Eich  
Netscape, May 1995

Mocha  
↓  
LiveScript  
↓  
JavaScript

"JS had to 'look like Java' only less so, be Java's dumb kid brother or boy-hostage sidekick. Plus, I had to be done in ten days or something worse than JS would have happened."

--Brendan Eich

# ECMA-262 TC39

JavaScript History

ECMA-262 (ES3)  
December 1999

ECMA-262 (ES4)

JavaScript History

ECMA-262 (ES5)  
December 2009

ECMA-262 (ES5.1)  
June 2011

“use strict”;  
Function#bind()  
Object.create()  
Object.defineProperty()  
Object.keys()  
get:, set:  
Array#forEach()  
...

# ECMA-262 (ES6) aka “Harmony”

\*mid 2014?

class Foo {...}  
export, import, module  
function foo(a, b=3) {...}  
function foo(a, ...r) {...}  
for-of  
Map, Set, WeakMap, Proxy  
Typed Arrays

...



# Agenda

## JavaScript Syntax

- Primitive Types
- Natives
- Coercion
- Reference vs. Value
- Surprises

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

```
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)

```
1
2
3  typeof null;    // "object" ... WAT!?
4
5
```

## 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"
```

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

Primitive Types: special values

# Negative Zero

```
1 var foo = 0 / -3;                                // true ... yay?
2 foo === -0;                                     // true ... doh!
3 foo === 0;                                      // true ... grr!
4 0 === -0;                                       // true ... sigh
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!
```

# NaN

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

Primitive Types: special values

# Nan

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

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?

Natives

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

Native Functions?

Natives

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

## Native Constructors

# 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"
```

```
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!
```

```
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

Explicit

Implicit

Coercion

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

Explicit Coercion

string       $\longleftrightarrow$       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

\* —> boolean

Explicit Coercion

# Falsy

""  
0, +0, -0  
null  
NaN  
false  
undefined

# Truthy

"foo"  
23  
{ a:1 }  
[1,3]  
true  
function(){..}

...

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 }
```

Implicit: happens as  
a side effect of some  
other operation

Implicit: *evil?*

Implicit Coercion

```
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!?
```

```
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"; // "foo"
15 baz;
```

```
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 }
```

```
1 "0" == null;           // false
2 "0" == undefined;    // false
3 "0" == false;         // true -- OOPS!!!
4 "0" == NaN;          // false
5 "0" == 0;             // true
6 "0" == "";            // false
7
8 false == null;        // false
9 false == undefined;   // false
10 false == NaN;         // false
11 false == 0;           // true -- OOPS!!!
12 false == "";          // true -- OOPS!!!
13
14 "" == null;           // false
15 "" == undefined;      // false
16 "" == NaN;            // false
17 "" == 0;               // true -- OOPS!!!
18
19 0 == null;            // false
20 0 == undefined;       // false
21 0 == NaN;             // false
```

## Implicit Coercion

```
1 [] == ![]; // true :(
2
3 [] == false; // true :(
4
5 [] == ""; // true :(
6
7 0 == ""; // true :(
8
9 !!"123"; // true :)
10 "123" == true; // false :(
```

# Ask Yourself:

1. Can the value ever be 0 or ""?
2. Can the value ever be [] or false?

Primitive  $\longleftrightarrow$  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

"Any sufficiently advanced technology  
is indistinguishable from magic."

--Arthur C. Clarke

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

--many devs

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

--many devs

== VS. ===

Implicit Coercion

**== checks value**

**==== checks value and type**

**== allows coercion**

**==== disallows coercion**

```
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 }
```

## Implicit Coercion: == vs. ===

but... but...

Implicit Coercion: == vs. ===

```
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 }
```

Implicit Coercion: == OK sometimes?

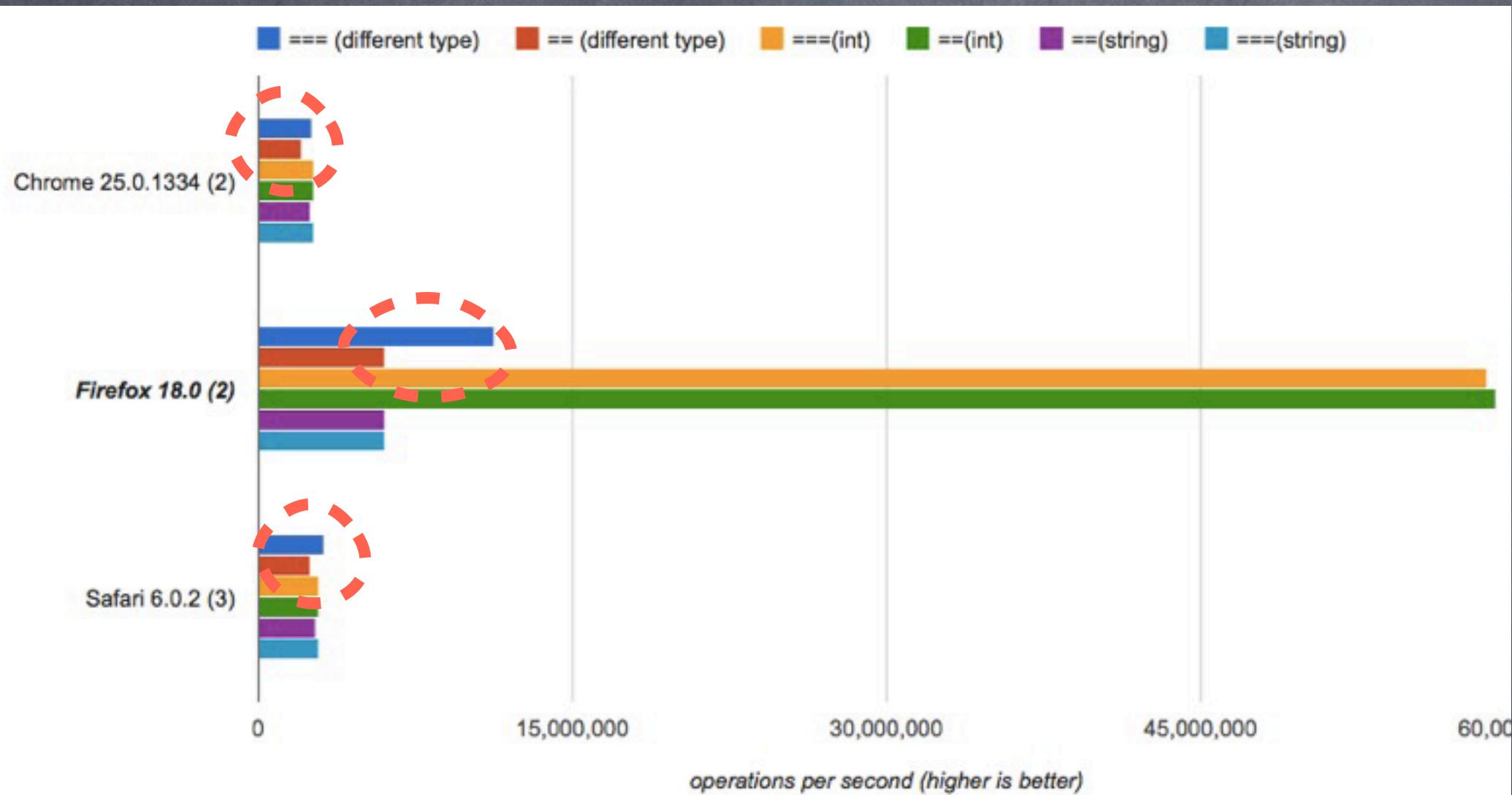
```
1 var foo;  
2 if (foo == null) { // yup!  
3     alert("Thanks!");  
4 }  
5  
6 foo = null;  
7 if (foo == null) { // yup  
8     alert("Thanks, again!");  
9 }  
10  
11 foo = false;  
12 if (foo == null) { // nope!  
13     alert("Phew!");  
14 }
```

Implicit Coercion: == OK sometimes?

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

Implicit Coercion: == vs. ===

<http://jsperf.com/triple>equals-vs-double>equals/5>



Implicit Coercion: == vs. === performance



- 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!

Implicit Coercion: `==` vs. `=====`

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

Implicit Coercion: == vs. ===

#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.

<http://jscoercion.qfox.nl/>

<http://webreflection.blogspot.com/2010/10/javascript-coercion-demystified.html>

Coercion

(exercise #1: 5min)

Reference vs. Value

Variables that hold simple primitives (string, number, boolean, undefined, null) are held **by value**.

Assignments are **by-copy**.

Variables that hold anything else are held **by reference**.

Assignments are by-reference.

```
1 function foo(obj) {  
2     obj.a = "foo";  
3     obj.b = [1,2,3];  
4     obj = { a: "baz", b: [4,5,6] };  
5 }  
6  
7 var o = { a: "bar", b: [7,8,9] };  
8 var a = o.a;  
9 var b = o.b;  
10  
11 foo(o);  
12  
13 o.a;          // "foo"  
14 o.b;          // [1,2,3]  
15 a;            // "bar"  
16 b;            // [7,8,9]
```

```
1 var obj = { b: [1,2,3] };  
2 var b = obj.b;  
3  
4 obj.b; // [1,2,3]  
5 b; // [1,2,3]  
6  
7 obj.b = null;  
8  
9 obj.b; // null  
10 b; // [1,2,3]  
11  
12 b = null;  
13  
14 obj.b; // null  
15 b; // null
```

<http://wtfjs.com>

Surprises

```
1
2  parseInt("08");           // 0 ... WAT!?
3
4  parseInt(1/0,19);         // 18 ... WAT!?
5
```

```
1 function foo(bar) {  
2     return  
3         bar + 2;  
4 }  
5  
6 foo(2);      // undefined ... WAT!?
```

```
1 42.; // valid syntax!
2
3 (42).toFixed(2); // "42.00"
4
5 42.toFixed(2); // SyntaxError
6
7 42.888.toFixed(2); // "42.89"
8
9 42..toFixed(2); // "42.00"
```

```
1 function hello(what) {  
2     what = "world";  
3     return "Hello, " + arguments[0] + "!";  
4 }  
5  
6 hello("friends"); // Hello, world!
```

```
1 // false -- hmmm...
2 String("abc") instanceof String;
3
4 // true
5 (new String("abc")) instanceof String;
6
7 // true -- wait, WAT! ?
8 String("abc") == (new String("abc"));
```

```
1  
2 3 > 2 > 1;           // false ... WAT!?  
3
```

```
1
2 Array(5).join("wat" - 1) + " Batman!";
3 // NaNNaNNaNNaN Batman!
4
```

```
1 Array(3).length === 3; // true
2
3 Array(3)[0] === Array(3)[1] ===
4     Array(3)[2] === undefined; // true
5
6 [undefined,undefined,undefined].toString(); // ,,
7 Array(3).toString(); // ,,
8
9 [undefined,undefined,undefined].join("+"); // ++
10 Array(3).join("+"); // ++
11
12 [undefined,undefined,undefined].map(
13     function(){ return "+" }
14 ); // ["+","+","+"]
15 Array(3).map(
16     function(){ return "+" }
17 ); // [] !?!?!
18
19 Array(3); // [] or "[undefined x 3]" (in Chrome console)
```

```
1 Array.apply(null,Array(3)); // [undefined,undefined,undefined]
2
3 Array.apply(null,Array(3)).map(
4     function(){ return "+" }
5 ); // ["+","+","+"]
6
7 Array.apply(null,{ length: 3 }).map(
8     function(){ return "+" }
9 ); // ["+","+","+"]
```

# Surprises

Please use JavaScript  
responsibly.

If you're going to write  
JavaScript, appreciate it.



# Agenda

## Optimizing DOM & Events

- CSS Selectors
- DOM Management
- Event Management

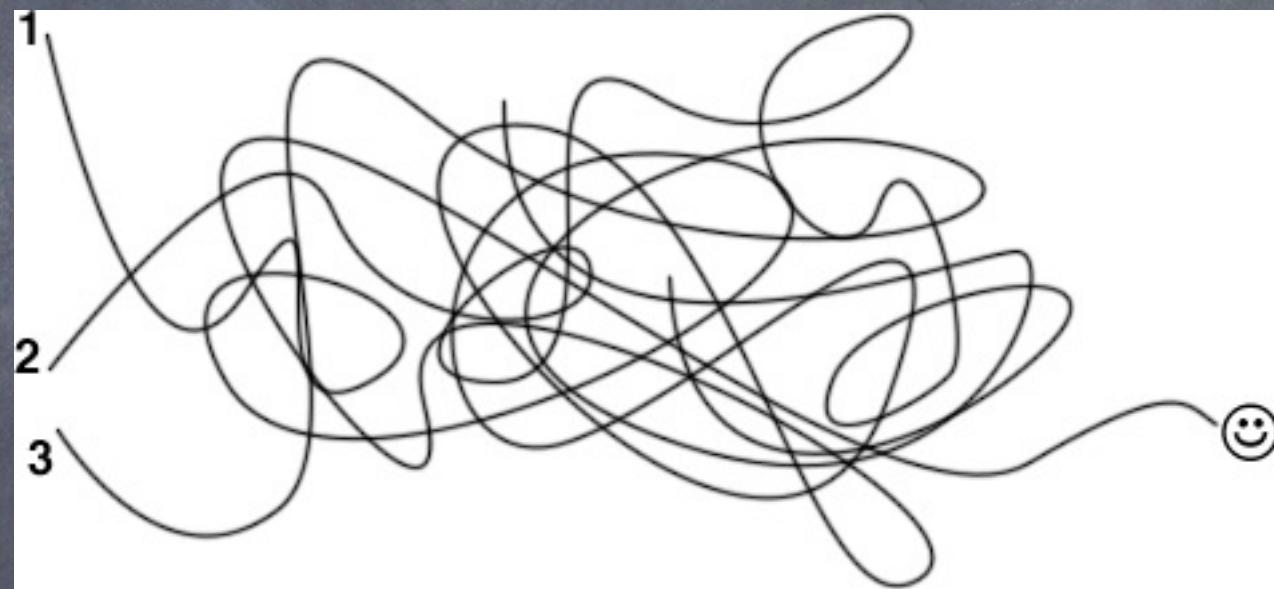
```
1 <ul>
2     <li id="foo">
3         <p>Bond, James Bond</p>
4     </li>
5     <li>
6         <p>Never say never again</p>
7     </li>
8     <li class="bar">
9         <p>Shaken, not stirred</p>
10    </li>
11    <li class="bar">
12        <p>Hello, Moneypenny</p>
13    </li>
14 </ul>
```

```
1 #foo { ... }  
2  
3 li.bar { ... }  
4  
5 #foo ~ li.bar > p { ... }
```

```
1 $("#foo ~ li.bar > p")
2 .addClass("highlighted")
3 .attr({
4     title: "Look at me!"
5 });
```

Left-to-Right  
Right-to-Left

<http://csswizardry.com/2011/09/writing-efficient-css-selectors/>



CSS Selectors

```
1 $("#foo ~ li.bar > p")
2 .addClass("highlighted")
3 .attr({
4     title: "Look at me!"
5 });
```

CSS Selectors: under-specified?

```
1 $("ul li#foo ~ li.bar > p")  
2 .addClass("highlighted")  
3 .attr({  
4     title: "Look at me!"  
5 });
```

```
1 var $bars = $("#foo ~ li.bar");
2
3 // later
4 $("> p", $bars)
5 .addClass("highlighted")
6 .attr({
7   title: "Look at me!"
8 });
```

CSS Selectors: jQuery “context”

```
1 var $bars = $("#foo ~ li.bar");
2
3 // later
4 $bars.children("p")
5 .addClass("highlighted")
6 .attr({
7   title: "Look at me!"
8 }));
```

```
1 var $bars = $("#foo ~ li.bar");
2
3 // later
4 $bars.children("p")
5 .addClass("highlighted")
6 .attr({
7   title: "Look at me!"
8 }));
```

```
1 var $bars = $("#foo ~ li.bar");
2
3 // later
4 $bars.children("p").each(function(){
5   $(this)
6     .addClass("highlighted")
7     .attr({
8       title: "Look at me!"
9     });
10});
```

# Quiz

```
1 <p id="p1">
2 Hello <em>world</em>!
3 </p>
4 <p class="p">
5 <strong>Now is the <em>time</em></strong>...
6 </p>
7 <script>
8 // find only these <em>'s
9 ??? .each(function(){
10     $(this)
11         .addClass("highlighted")
12         .attr({
13             title: "Look at me!"
14         });
15 });
16 </script>
```

# Quiz

```
1 <p id="p1">
2 Hello <em>world</em>!
3 </p>
4 <p class="p">
5 <strong>Now is the <em>time</em></strong>...
6 </p>
7 <script>
8 // find only these <em>'s
9 $("#p1 > em, .p > strong > em").each(function(){
10     $(this)
11         .addClass("highlighted")
12         .attr({
13             title: "Look at me!"
14         });
15 });
16 </script>
```

(exercise #2: 10min)

DOM: toll road

DOM Management

```
1 for (var i=1; i<6; i++) {  
2   $("h" + i)  
3     .addClass("header");  
4 }
```

```
1 var h = [];
2 for (var i=1; i<6; i++) {
3     h.push("h" + i);
4 }
5
6 $(h.join(","))
7 .addClass("header");
```

```
1 var mobsters = ["Lou","Frankie","Max"];
2 var $li;
3
4 for (var i=0; i<mobsters.length; i++) {
5   $li = $("<li></li>")
6     .text(mobsters[i]);
7
8   $("#hit_list").append($li);
9 }
```

```
1 var mobsters = ["Lou","Frankie","Max"];
2 var $li;
3 var $hit_list = $("#hit_list");
4
5 for (var i=0; i<mobsters.length; i++) {
6   $li = $("<li></li>")
7     .text(mobsters[i]);
8
9   $hit_list.append($li);
10 }
```

```
1 var mobsters = ["Lou","Frankie","Max"];
2 var $li;
3 var $hit_list = $("#hit_list");
4
5 for (var i=0; i<mobsters.length; i++) {
6   $li = $("<li></li>")
7   .text(mobsters[i]);
8
9   $hit_list.append($li);
10 }
```

```
1 var mobsters = ["Lou", "Frankie", "Max"];  
2 var $li;  
3 var $lis;  
4 var $hit_list = $("#hit_list");  
5  
6 for (var i=0; i<mobsters.length; i++) {  
7     $li = $("<li></li>")  
        .text(mobsters[i]);  
8  
9  
10    if (!$lis) { $lis = $($li); }  
11    else { $lis = $lis.add($li); }  
12 }  
13  
14 $hit_list.append($lis);
```

DOM Management: insertions

```
1 var mobsters = ["Lou","Frankie","Max"];
2 var lis = [];
3 var $hit_list = $("#hit_list");
4
5 for (var i=0; i<mobsters.length; i++) {
6   lis.push("<li>" + mobsters[i] +
7     "</li>"); ...
8 }
9
10 $hit_list.html(lis.join(""));
```

DOM Management: insertions

Event Management

```
1 var mobsters = ["Lou","Frankie","Max"];
2 var $li;
3 var $hit_list = $("#hit_list");
4
5 for (var i=0; i<mobsters.length; i++) {
6   $li = $("<li></li>")
7   .text(mobsters[i]);
8
9   $hit_list.append($li);
10 }
```

## Event Management

```
1 var $hit_list = $("#hit_list");
2
3 $hit_list.children("li")
4 bind("click", function(evt){
5     var who = $(evt.target).text();
6     alert("You just shot " + who);
7});
```

## Event Management

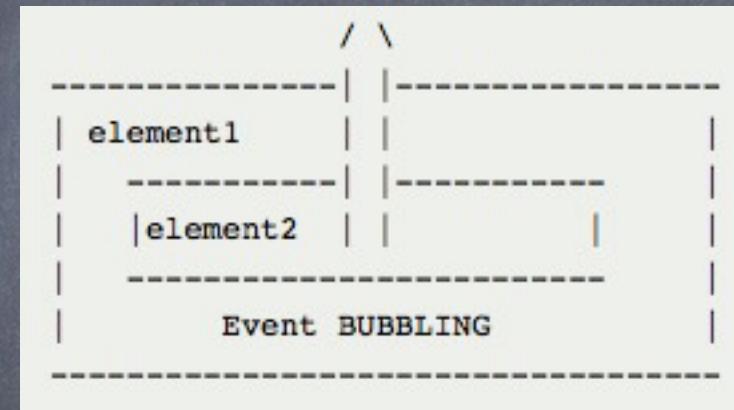
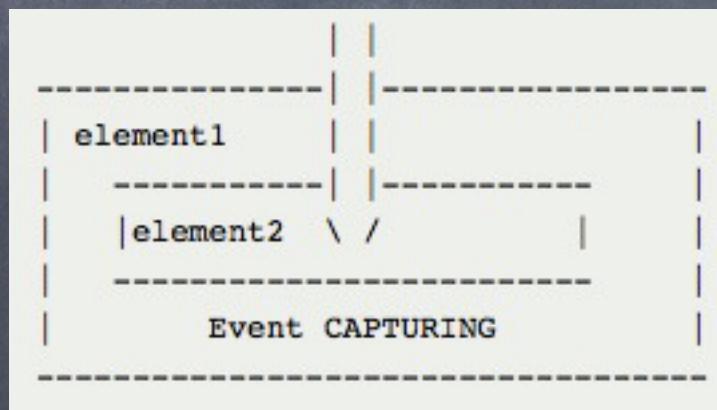
# Event Propagation

Capturing

Bubbling

Event Management

[http://www.quirksmode.org/js/events\\_order.html](http://www.quirksmode.org/js/events_order.html)



Event Management

```
1 var $hit_list = $("#hit_list");
2
3 $hit_list
4 .bind("click",function(evt){
5     var who = $(evt.target).text();
6     alert("You just shot " + who);
7 });
```

```
1 var $hit_list = $("#hit_list");
2
3 $hit_list
4 .bind("click",function(evt){
5     if ($(evt.target).is("li")) {
6         var who = $(evt.target).text();
7         alert("You just shot " + who);
8     }
9 });
```

```
1 var $hit_list = $("#hit_list");
2
3 $hit_list
4 .on("click", "> li", function(evt){
5     var who = $(evt.target).text();
6     alert("You just shot " + who);
7 });
```

## Event Management: delegation

focus

change

Event Management: capturing

```
1 var $hit_list = $("#hit_list");
2
3 $hit_list
4 .bind("focus",function(evt){
5     alert("Watch out!");
6 });
```

Event Management: capturing

```
1 // cheat :)  
2 $hit_list.find("li").append(  
3     $("<input>")  
4 );  
5  
6 $hit_list[0].addEventListener(  
7     "focus",  
8     function(evt){  
9         console.log("Watch out!");  
10    },  
11    /*capturing*/true  
12 );
```

Event Management: capturing

<http://fuelyourcoding.com/jquery-events-stop-misusing-return-false/>

Event Management: canceling

```
1 $("#nav").on(  
2     "click",  
3     "> a",  
4     function(evt){  
5         // do something useful!  
6  
7         // prevent default  
8         return false;  
9     }  
10 );
```

Event Management: canceling

```
1 $("#nav").on(  
2     "click",  
3     "> a",  
4     function(evt){  
5         // do something useful!  
6  
7         // prevent default  
8         evt.preventDefault();  
9     }  
10 );
```

Event Management: canceling

```
1 $("#nav").on(  
2     "click",  
3     "> a",  
4     function(evt){  
5         // do something useful!  
6  
7         // prevent propagations  
8         evt.stopPropagation();  
9         evt.stopImmediatePropagation();  
10    }  
11 );
```

Event Management: canceling

```
1 $("#nav")[0].addEventListener(  
2     "click",  
3     function(evt){  
4         evt.preventDefault();  
5         evt.stopPropagation();  
6         evt.stopImmediatePropagation();  
7     },  
8     /*capturing*/true  
9 );
```

```
1 $("#nav")[0].addEventListener(  
2   "focus",  
3   function(evt){  
4     console.log("cancelable: " +  
5       evt.cancelable // false :(  
6     );  
7   },  
8   /*capturing=*/true  
9 );
```

Event Management: canceling

(exercise #3: 20min)

```
1 $("#nav").on(  
2   "disabled", // custom event!  
3   "> a",  
4   function(evt){  
5     $(evt.target)  
6       .addClass("disabled");  
7   }  
8 );  
9  
10 // later  
11 $("#nav > a").trigger("disabled");
```

Event Management: custom DOM events

<https://github.com/hij1nx/EventEmitter2>

<https://github.com/creationix/eventemitter-browser>

```
1 var EVT = new EventEmitter2();
2
3 // later
4 EVT.on("foobar", function(){
5     console.log("foobar fired!");
6 });
7
8 // even later
9 EVT.emit("foobar");
```



# 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?

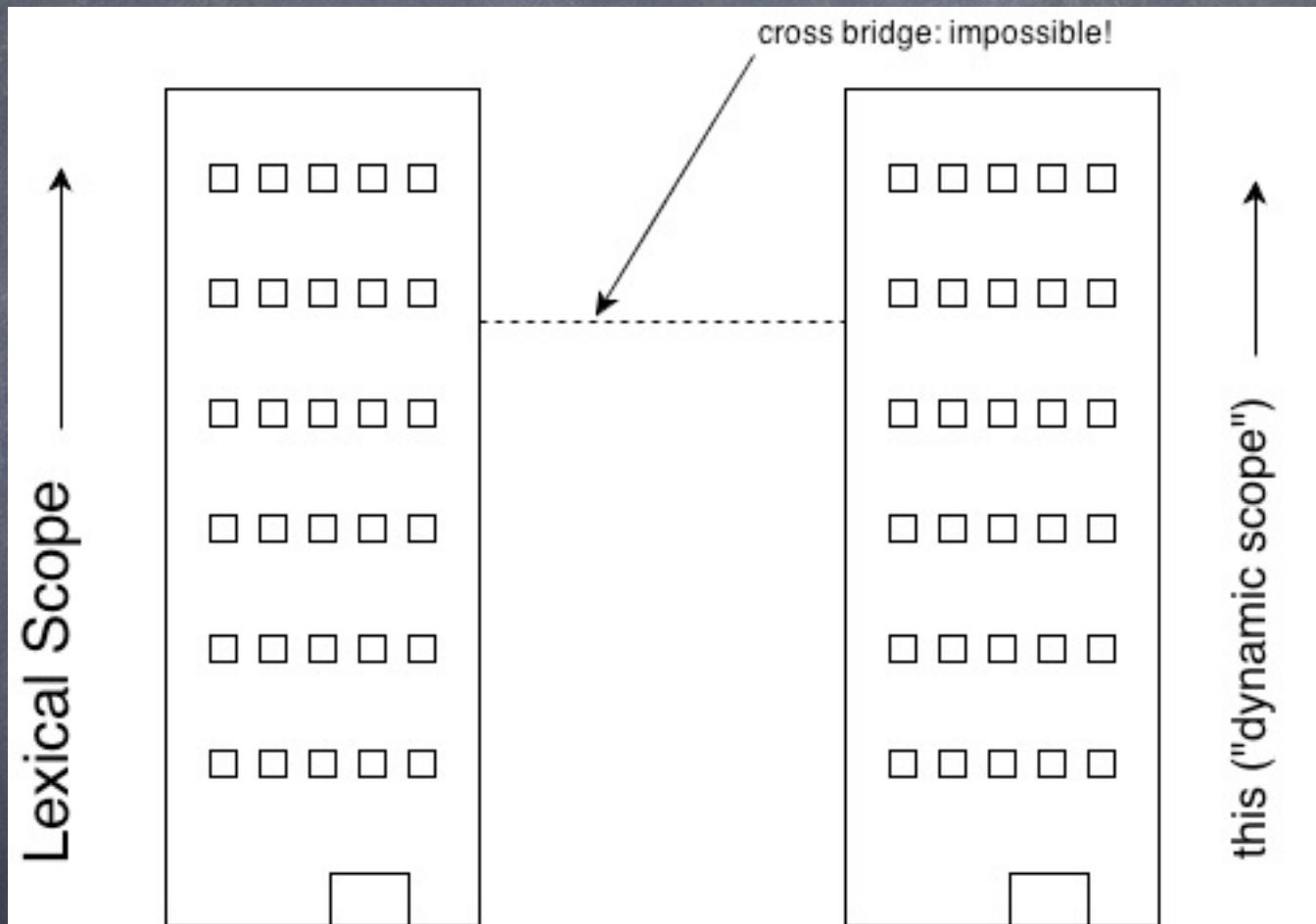
```
1 var foo;  
2  
3 try {  
4     foo.length;  
5 }  
6 catch (err) {  
7     console.log(err); // TypeError  
8 }  
9  
10 console.log(err); // ReferenceError
```

Scope: block scope?

lexical scope

dynamic scope

Scope



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

this

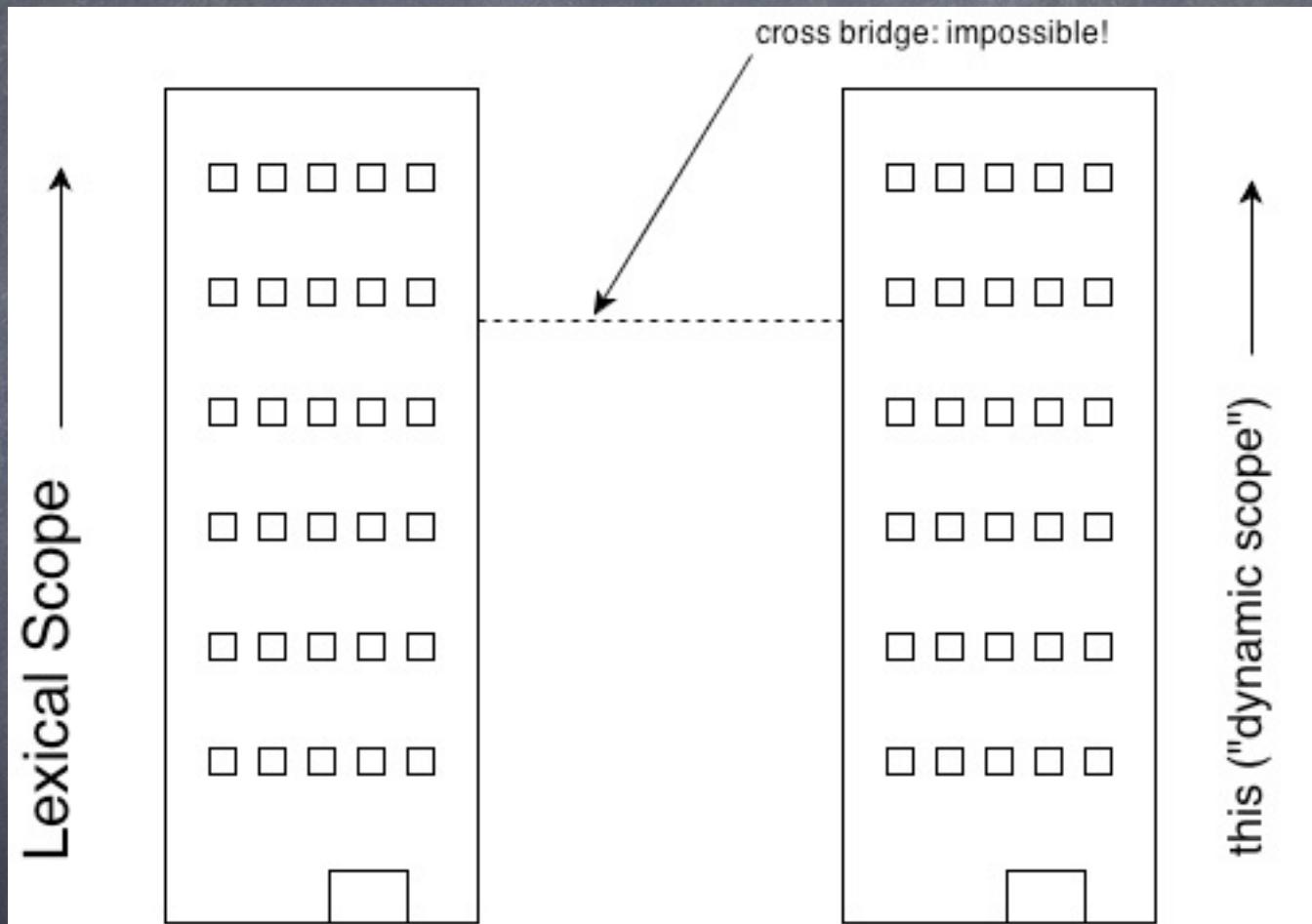
Every function, while executing, has a reference to its current execution context, called **this**.

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

[https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global\\_Objects/Function/bind](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Function/bind)

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

this: hard binding

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

this: new

```
1 function something() {
2     this.hello = "hello";
3     console.log(this.hello, this.who);
4 }
5
6 var who = "global", foobar, bazbam,
7     obj1 = { who: "obj1", something: something },
8     obj2 = { who: "obj2" };
9
10 something(); // "hello" "global"
11 console.log(hello); // "hello"           <-- OOPS!!!
12
13 obj1.something(); // "hello" "obj1"
14 console.log(obj1.hello); // "hello"
15
16 obj1.something.call(obj2); // "hello" "obj2"
17 console.log(obj2.hello); // "hello"
18
19 foobar = something.bind(obj2);
20 foobar(); // "hello" "obj2"
21 foobar.call(obj1); // "hello" "obj2"
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 #5: 20min)

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



# Agenda

## Object-Orienting

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

# Information hiding

Encapsulation

# Principle of Least Exposure

Encapsulation

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

Encapsulation

private  
protected  
public

Encapsulation

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

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

Encapsulation: this'ification

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

Encapsulation: this'ification

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

Encapsulation: this'ification

# OO Design Patterns

# Singleton

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

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

# 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

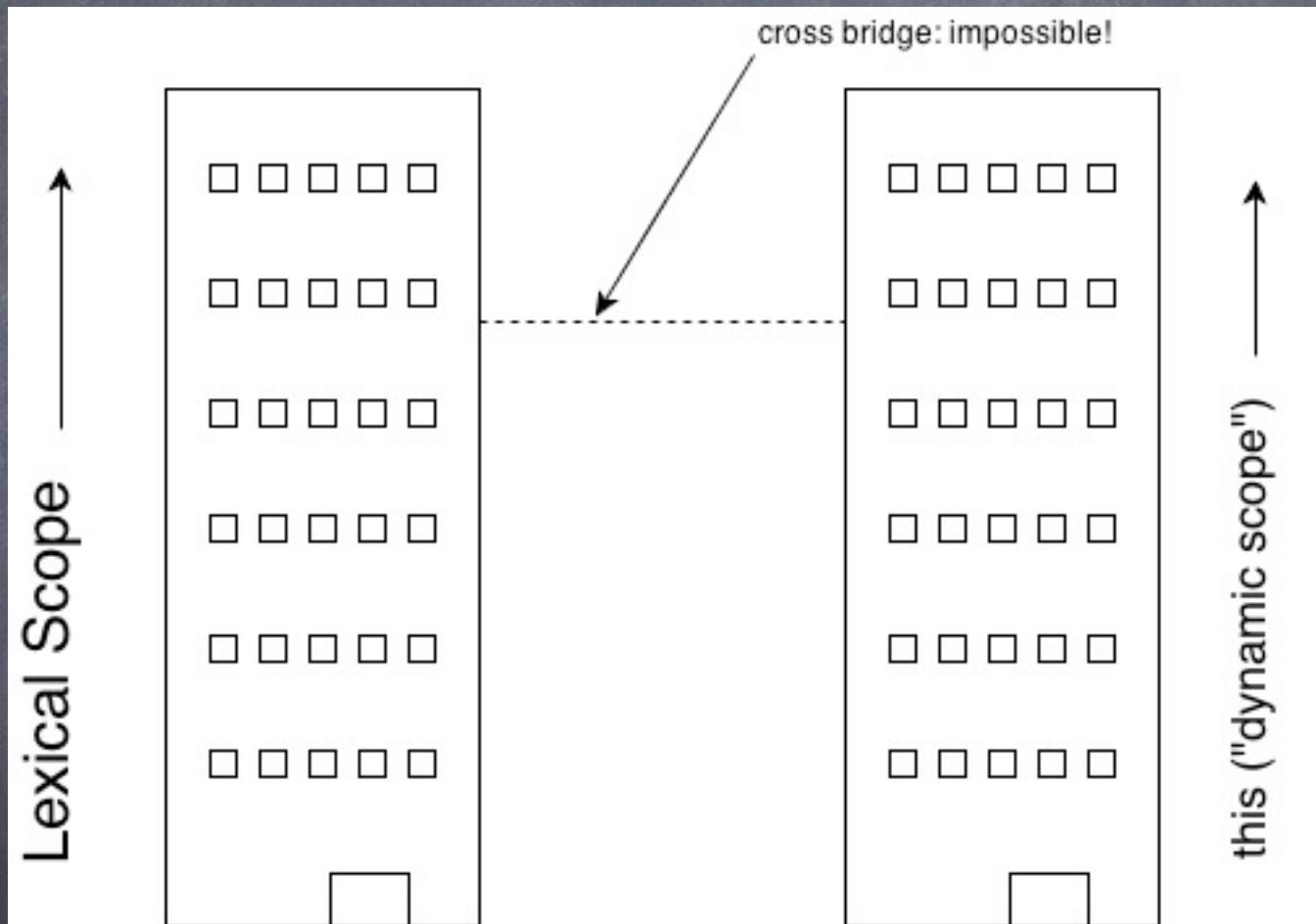
```
1 function Foo(who) {  
2     this.me = who;  
3 }  
4 Foo.prototype.identify = function() {  
5     return "I am " + this.me;  
6 };  
7  
8 var a1 = new Foo("a1");  
9 var a2 = new Foo("a2");  
10  
11 a2.speak = function() {  
12     alert("Hello, " + this.identify() + ".");  
13 };  
14  
15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

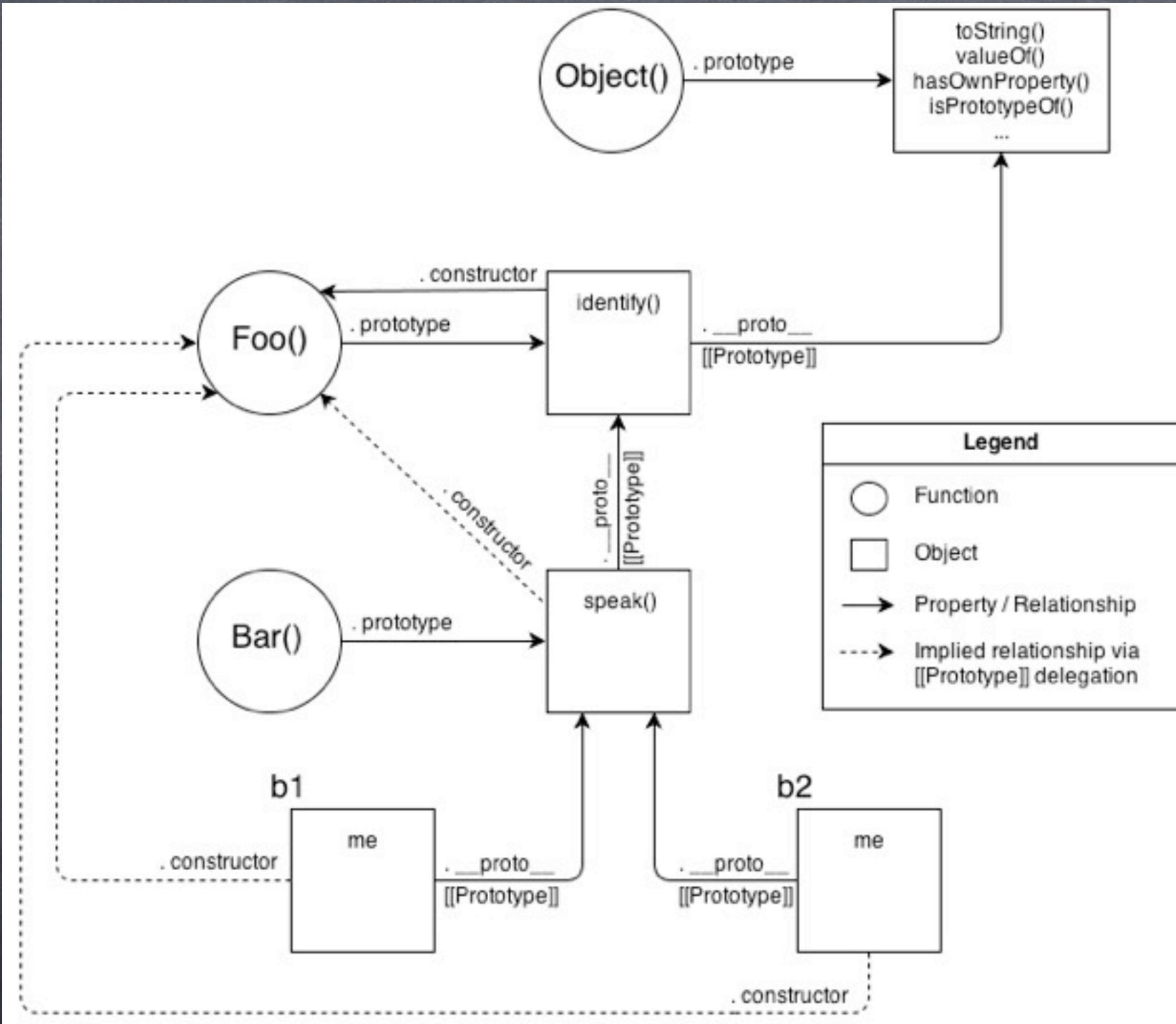
prototype



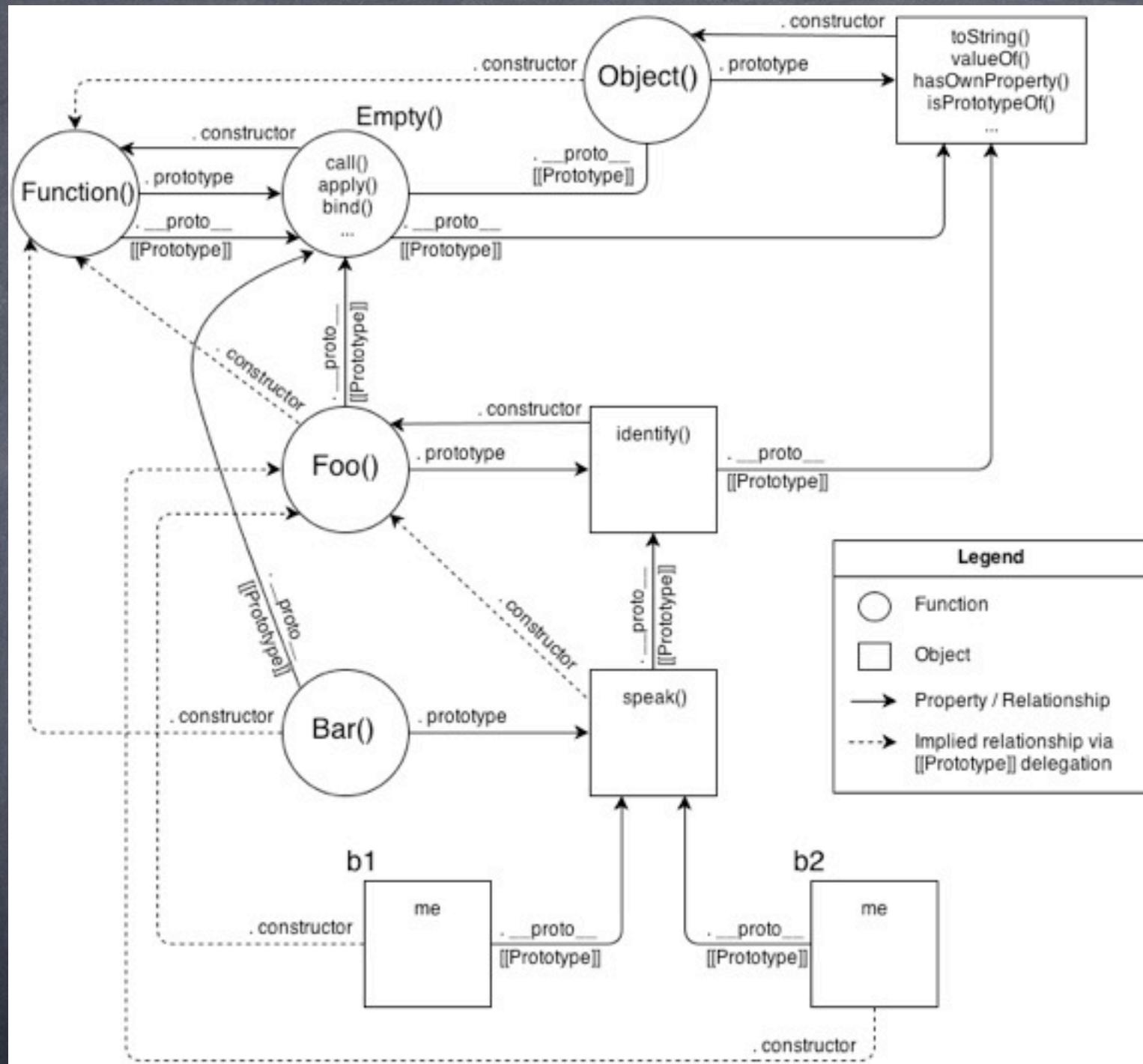
prototype

```
1 function Foo(who) {  
2     this.me = who;  
3 }  
4 Foo.prototype.identify = function() {  
5     return "I am " + this.me;  
6 };  
7  
8 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

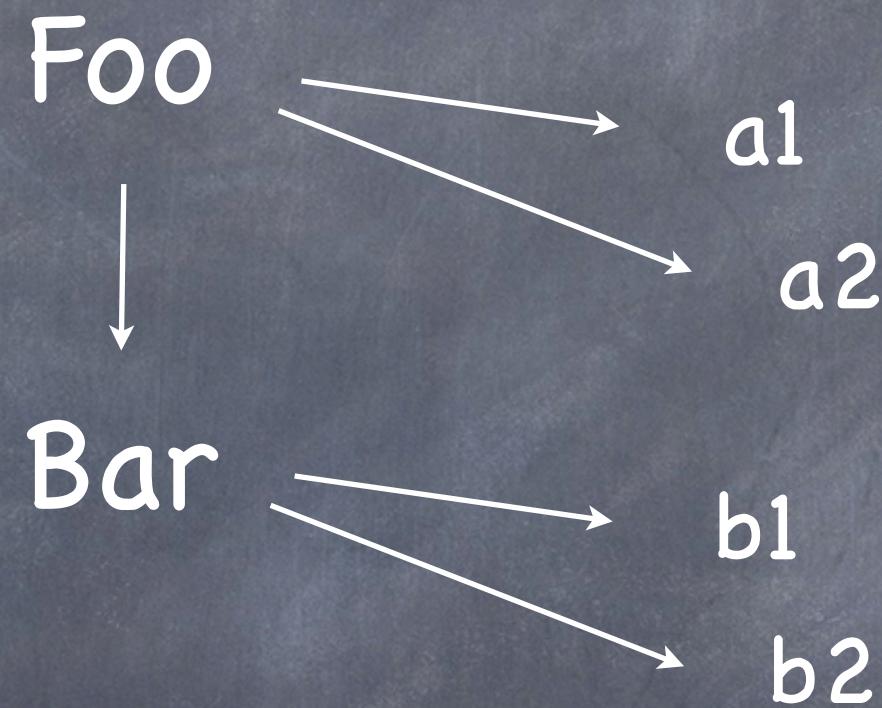
```
1 function Foo(who) {  
2     this.me = who;  
3 }  
4  
5 Foo.prototype.speak = function() {  
6     alert("Hello, I am " + this.me + ".");  
7 };  
8  
9 var a1 = new Foo("a1");  
10  
11 $("#speak").click(a1.speak);
```

prototype: this redux

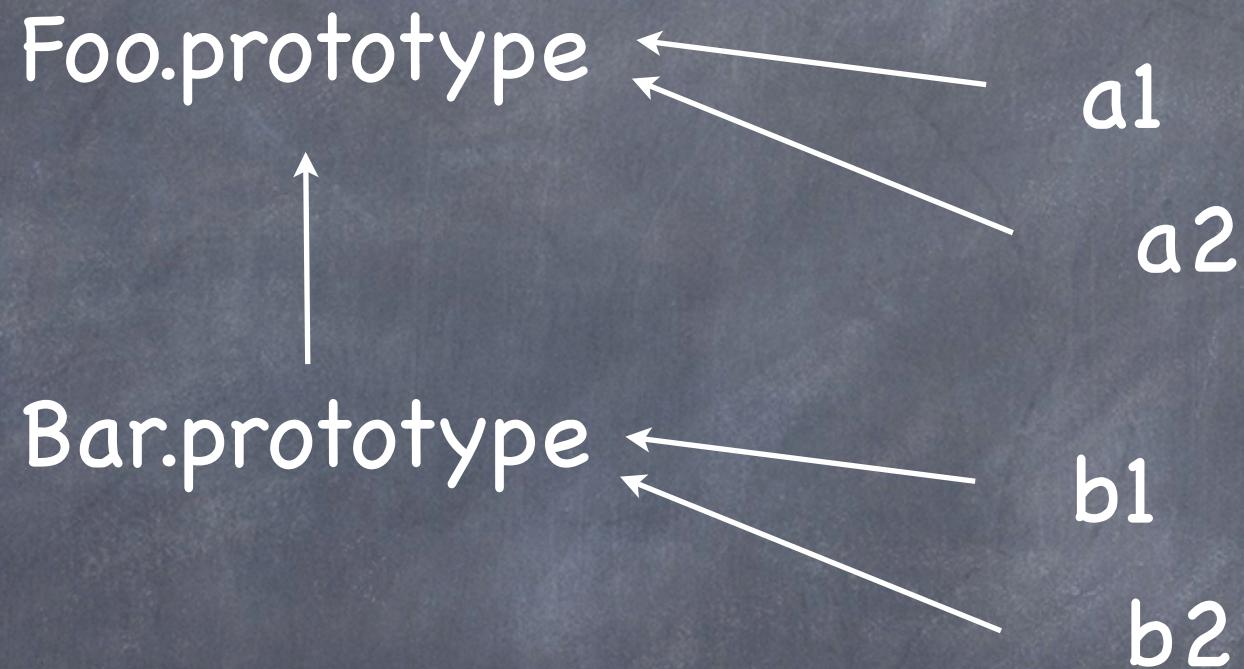
(exercise #7: 20min)

# Inheritance

00



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

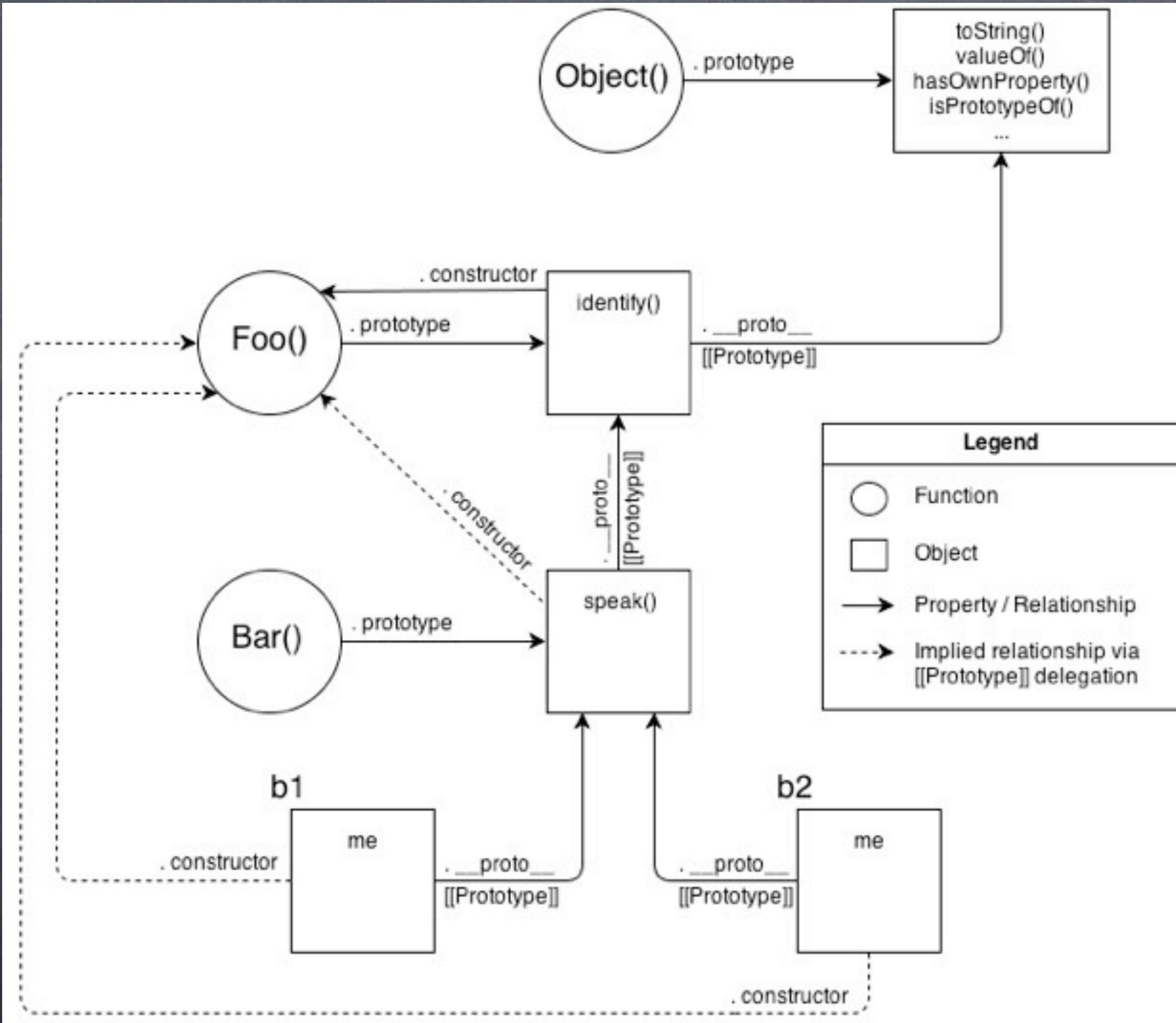
```
1 function Foo(who) {  
2     this.me = who;  
3 }  
4 Foo.prototype.identify = function() {  
5     return "I am " + this.me;  
6 };  
7  
8 var Bar = Object.create(Foo.prototype);  
9 Bar.init = function(who) {  
10     Foo.call(this, who);  
11 };  
12 Bar.speak = function() {  
13     alert("Hello,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

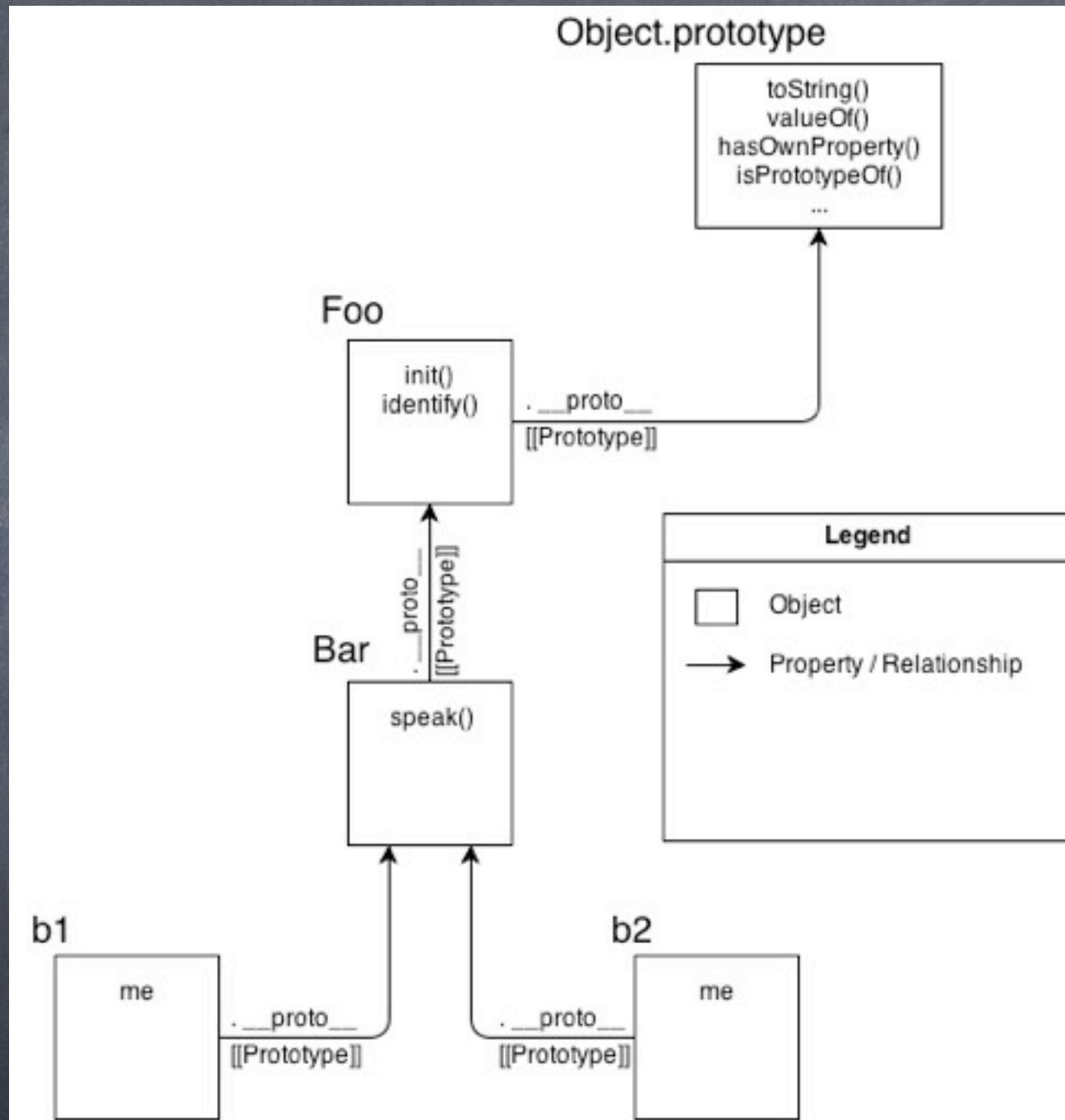
OLOO: delegated objects

```
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

```
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 Bar.init = function(who) {  
12     this.__proto__.init(who); // Uh oh!  
13 };  
14 Bar.speak = function() {  
15     alert("Hello, " + this.identify() + ".");  
16 };  
17  
18 var b1 = Object.create(Bar);  
19 b1.init("b1");  
20 b1.speak(); // alerts: "Hello, I am b1."
```

OLOO: relative polymorphism?

```
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 Bar.init = function(who) {  
12     Foo.init.call(this, who); // Ugly  
13 };  
14 Bar.speak = function() {  
15     alert("Hello, " + this.identify() + ".");  
16 };  
17  
18 var b1 = Object.create(Bar);  
19 b1.init("b1");  
20 b1.speak(); // alerts: "Hello, I am b1."
```

OLOO: relative polymorphism mixins! :(

<https://gist.github.com/getify/5297595>

OLOO: relative polymorphism...sorta

# 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 #8: 10min)



# Agenda

## UI Performance

- Why, How?
- Resource Loading
- UI Thread & GC
- jsPerf

performance

caveat lector

caveat utilitor

How web  
performance?

Web

Measurement

Perception

# Efficiency

Speed

Memory

# Performance

Benchmark

Optimize

“premature optimization is  
the root of all evil”

--Donald Knuth

“Programmers waste enormous amounts of time...worrying about the speed of **noncritical** parts of their programs...We should forget about small efficiencies, say about 97% of the time...Yet we should not pass up our opportunities in that **critical 3%**.”

--Donald Knuth

# Optimization

noncritical

critical

“noncritical path optimization  
is the root of all evil”  
--Donald Knuth

“**immature** optimization is  
the root of all evil”

--getify

# Mature Optimization

100ms  
↔  
1000ms

<http://www.useit.com/papers/responsetime.html>

“Death by a  
thousand papercuts”

Inefficiency is  
systemic

# The myth of the refactor

TCO

Performant by  
Default

# Resource Loading

# Preloading

Resource Loading

```
1 <link rel="prefetch" href="something.jpg">
2
3 <script>
4 var img = new Image();
5 img.src = "something.jpg";
6 </script>
7
8
```

# Lazy Loading On-demand Loading Postloading

Resource Loading

```
1 function scriptLoaded() {  
2     // yay!  
3 }  
4  
5 var scr = document.createElement("script");  
6 scr.src = "foobar.js";  
7 document.head.appendChild(scr);  
8  
9 scr.onload = scriptLoaded;  
10 scr.onreadystatechange = function() {  
11     if (scr.readyState === "loaded" || scr.readyState === "complete") {  
12         scriptLoaded();  
13     }  
14 }  
15
```

Parallel Loading  
preserve execution order

Resource Loading

```
1 function allScriptsLoaded() {
2     // yay!
3 }
4
5 function loadScript(source,done) {
6     var scr = document.createElement("script");
7     scr.src = source;
8     scr.async = false; // ----- LOOKEY LOOKEY!!
9     document.head.appendChild(scr);
10
11    if (done) {
12        scr.onload = done;
13        scr.onreadystatechange = function() {
14            if (scr.readyState === "loaded" || scr.readyState === "complete") {
15                done();
16            }
17        }
18    }
19 }
20
21 loadScript("jquery.js");
22 loadScript("foobar.js");
23 loadScript("baz.js",allScriptsLoaded);
```

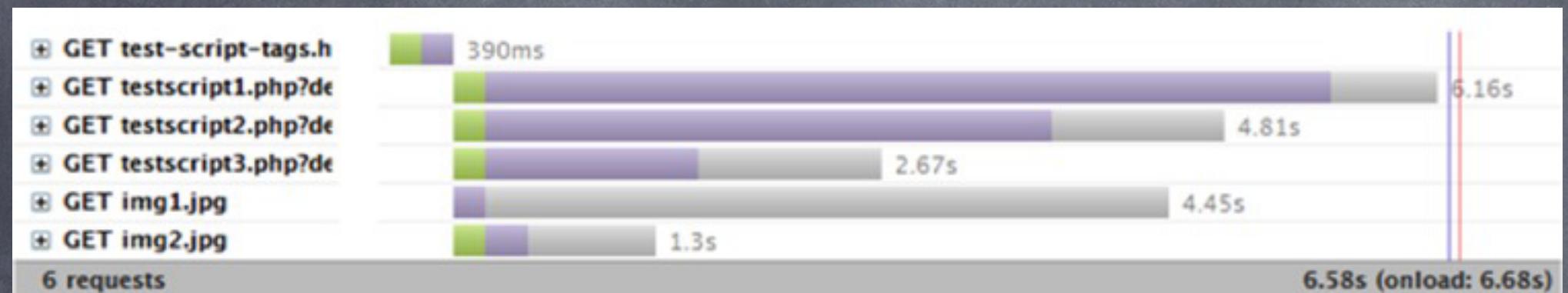
# Resource Loading

<http://LABjs.com/>

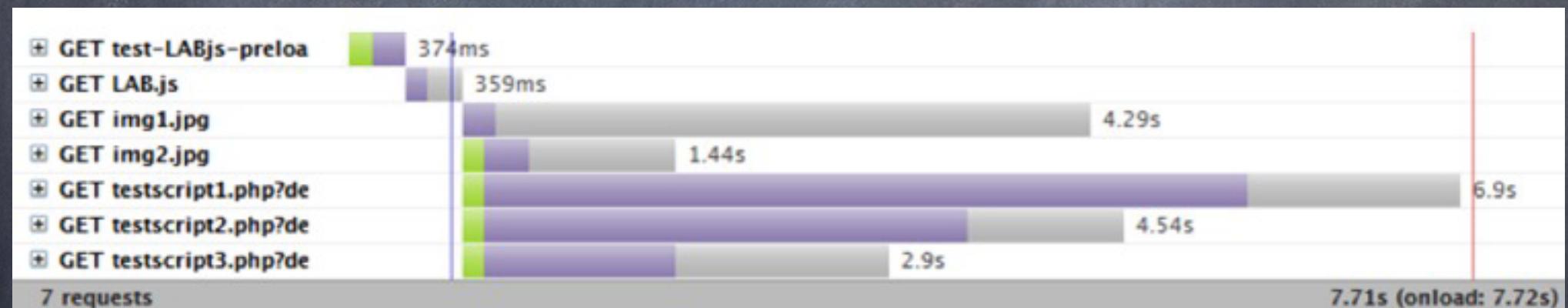
Resource Loading

# Why Dynamic?

Resource Loading



# <script>



# dynamic loading

# Resource Loading

# Single Threaded

UI Thread

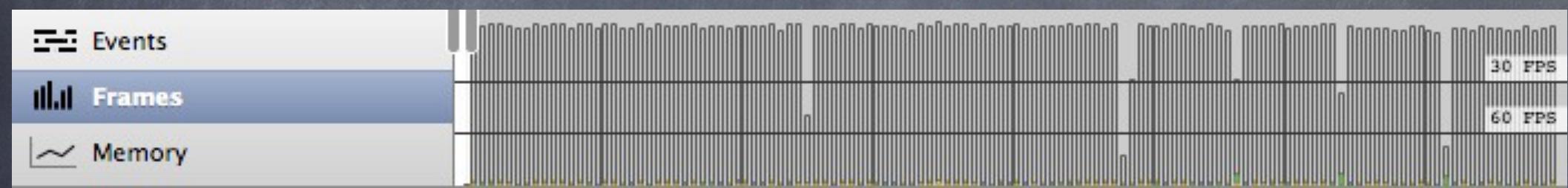
# Async vs. Parallel

UI Thread

# requestAnimationFrame

UI Thread

```
1 function updateElement() {
2     $elem.css({
3         left: x + "px",
4         top: y + "px"
5     });
6 }
7
8 var x, y;
9
10 function doAnimation() {
11     // calculate new x,y
12     updateElement();
13     requestAnimationFrame(doAnimation);
14 }
15
16 requestAnimationFrame(doAnimation);
17
```



UI Thread

# “Threaded” JavaScript

## Web Workers

UI Thread

```
1 var longThread = new Worker("long.js");
2
3 longThread.onmessage = function(evt) {
4     var answer = evt.data;
5     // do something with the answer
6 };
7
8 longThread.postMessage({
9     question: "..."
10 });
11
```

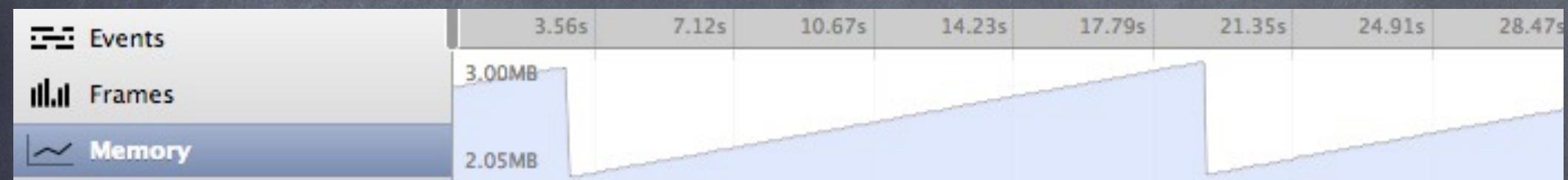
```
1 // long.js
2
3 function figureOutAnswer(question) {
4     // ... lots of work
5
6     return answer;
7 }
8
9 self.onmessage = function(evt) {
10    var question = evt.data.question;
11
12    // will take awhile!
13    var answer = figureOutAnswer(question);
14
15    // finally, send the answer back
16    self.postMessage(answer);
17 };
```

# Dynamic Memory Allocation

Garbage Collection

```
1 var obj = {  
2     // lots of data  
3 };  
4  
5 // later  
6  
7 obj = null;  
8  
9 // what now?  
10  
11
```

# Garbage Collection



# Garbage Collection

(exercise #11: 20min)

**Test runner**

Handlebars × 8,536 (37 samples)

**Stop running**

Testing in Firefox 18.0 on Mac OS X 10.7

	<b>Test</b>	<b>Ops/sec</b>
<b>Mustache 0.5.0 DEV (compiled)</b>	tpl.mustache(tpl.vars);	completed
<b>Kendo UI</b>	tpl.kendo(tpl.vars);	completed
<b>Kendo UI (no "with" block)</b>	tpl.kendo2(tpl.vars);	completed
<b>Handlebars</b>	tpl.handlebars(tpl.vars)	pending...
<b>Underscore</b>	tpl.underscore(tpl.vars);	pending...
<b>Resig Micro</b>	tpl.resig(tpl.vars)	pending

**grips (direct partial,  
pre-compiled)**

```
// NOTE: this variation of the previous grips test is only present to get around this  
performance "bug" in FF, otherwise the "compiled" test for grips would have been sufficient.  
// http://jsperf.com/functions-created-globally-or-via-new-function  
  
tpl.grips2( tpl.vars );
```

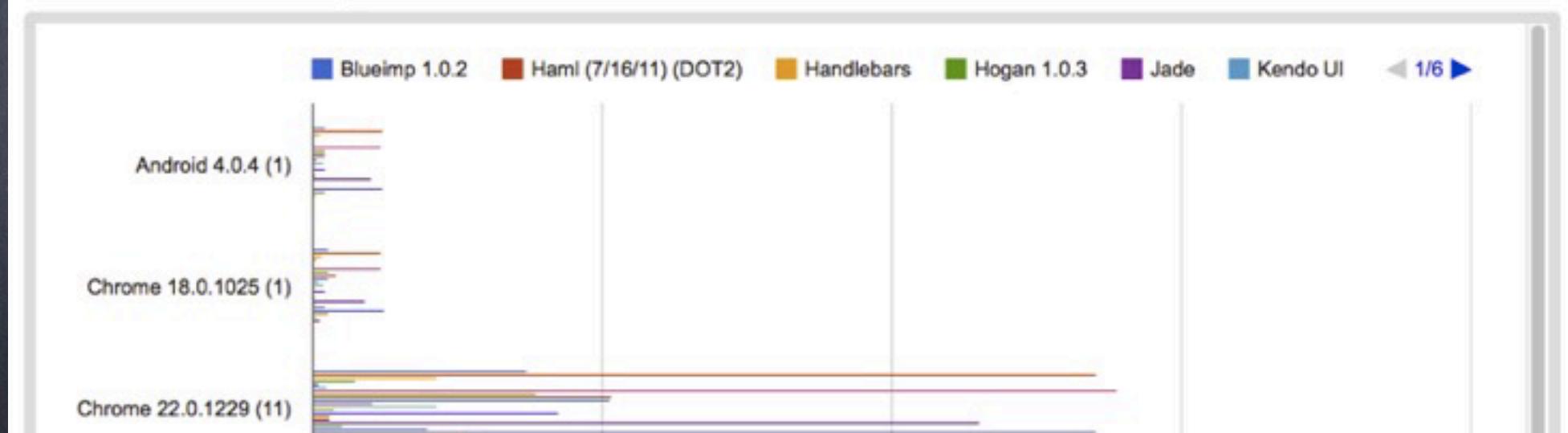
ready

### Compare results of other browsers

Chart type: [bar](#), [column](#), [line](#), [pie](#), [table](#)

Filter: [popular](#), [all](#), [desktop](#), [family](#), [major](#), [minor](#), [mobile](#), [prerelease](#)

## Browserscope



<http://jsperf.com>

jsPerf

“**immature** optimization is  
the root of all evil”

--getify



# 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"
```

## Async Patterns: generators

```
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

<https://github.com/getify/asynquence>

```
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

```
1 ASQ(2)
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});
```

## Async Patterns: CSP-style generators

# 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 #13: 15min)



Take a deep breath

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

Thanks!

Questions?