Javascript in Ten Minutes

Spencer Tipping

December 4, 2011

Contents

1 Introduction

This guide is for anyone who knows some Javascript but would like a quick[1] intro to its advanced features. It will be easier reading if you also know another functional language such as Ruby, Perl, Python, ML, Scheme, etc, since I don't really explain how first-class functions work.

2 Types

Javascript has nine types. They are:

1. Null – `null`. Chucks a wobbly if you ask it for any attributes; e.g. `null.foo` fails. Never boxed.[2]
2. Undefined – `undefined`. What you get if you ask an object for something it doesn't have; e.g. `document.nonexistent`. Also chucks a wobbly if you ask it for any attributes. Never boxed.
3. Strings – e.g. `'foo'`, `"foo"` (single vs. double quotation marks makes no difference). Sometimes boxed. Instance of `String` when boxed.
4. Numbers – e.g. `5`, `3e+10` (all numbers behave as floats – significant for division, but can be truncated by `x >>> 0`). Sometimes boxed. Instance of `Number` when boxed.
5. Booleans – `true` and `false`. Sometimes boxed. Instance of `Boolean` when boxed.
6. Arrays – e.g. `[1, 2, "foo", [3, 4]]`. Always boxed. Instance of `Array`.
7. Objects – e.g. `{foo: 'bar', bif: [1, 2]}`, which are really just hashtables. Always boxed. Instance of `Object`.
8. Regular expressions – e.g. `/foo\s*([bar]+)/`. Always boxed. Instance of `RegExp`.
9. Functions – e.g. `function (x) {return x + 1}`. Always boxed. Instance of `Function`.

The value `null` is actually almost never produced by Javascript. The only case you're likely to run across `null` is if you assign it somewhere (most of the time you'll get `undefined` instead – one notable exception is `document.getElementById`, which returns `null` if it can't find an element). Making sparing use of `undefined` and instead using `null` can make bugs much easier to track down.

[1] Longer than ten minutes, despite what the title says.

[2] Boxing is just a way of saying whether something has a pointer. A boxed type is a reference type, and an unboxed type is a value type. In Javascript, this has additional ramifications as well – see section 4.6.

3 Functions

Functions are first-class lexical closures,[3] just like `lambda`s in Ruby or `sub`s in Perl.[4] They behave pretty much like you'd expect, but there are several really cool things about functions and one really egregious disaster.

3.1 Variadic behavior (a cool thing)

Functions are always variadic.[5] Formal parameters are bound if they're present; otherwise they're `undefined`. For example:

```
(function (x, y) {return x + y}) ('foo')        // => 'fooundefined'
```

The arguments to your function can be accessed in a first-class way, too:

```
var f = function () {return arguments[0] + arguments[1]};
var g = function () {return arguments.length};
f ('foo')                    // => 'fooundefined'
g (null, false, undefined)   // => 3
```

The `arguments` *keyword is not an array!* It just looks like one. In particular, doing any of these will cause problems:

```
arguments.concat ([1, 2, 3])
[1, 2, 3].concat (arguments)
arguments.push ('foo')
arguments.shift ()
```

To get an array from the `arguments` object, you can say `Array.prototype.slice.call (arguments)`. As far as I know that's the best way to go about it.

3.2 Lazy scoping (a cool thing)

Internally, functions use a lexical scoping chain. However, the variables inside a function body aren't resolved until the function is called. This has some really nice advantages, perhaps foremost among them self-reference:

```
var f = function () {return f};
f () === f              // => true
```

[3] First-class in the sense that you can pass them around as values at runtime. You can't reliably introspect them, however, because while you can obtain their source code via `toString` you won't be able to access the values they close over.

[4] Note that block scoping isn't used – the only scopes that get introduced are at function boundaries.

[5] The number of arguments a function accepts is referred to as its *arity*. So a unary function, which is monadic, takes one, a binary function, which is dyadic, takes two, etc. A function that takes any number of arguments is said to be variadic.

Tidbit of pathology: An important consequence of lazy scoping is that you can create functions that refer to variables that might never exist. This makes Javascript very difficult to debug. The good part is that Javascript can be made to support syntactic macros via the `toString` method:

```
var f = function () {return $0 + $1};
var g = eval (f.toString ().replace (/\$(\d+)/g,
          function (_, digits) {return 'arguments[' + digits + ']'}));
g (5, 6)        // => 11 (except on IE)
```

Theoretically by extending this principle one could implement true structural macros, operator overloading, a type system,[6] or other things.

3.3 The meaning of `this` (the egregious disaster)

One would think it is a simple matter to figure out what `this` is, but it's apparently quite challenging, and Javascript makes it look nearly impossible. Outside of functions (in the global scope, that is), the word `this` refers to the *global object*, which is `window` in a browser. The real question is how it behaves inside a function, and that is determined entirely by how the function is called. Here's how that works:

1. If the function is called alone, e.g. `foo(5)`, then inside that function's body the word `this` will be equivalent to the global object.

2. If the function is called as a method, e.g. `x.foo(5)`, then inside that function's body the word `this` refers to the object, in this case `x`.

3. If the function starts off as a method and then is called alone:

   ```
   var f = x.foo;
   f (5);
   ```

   then `this` will be the global object again. Nothing is remembered about where `f` came from; it is all determined right at the invocation site.

4. If the function is invoked using `apply` or `call`, then `this` points to whatever you set it to (unless you try to set it to `null` or `undefined`, in which case it will be the global object again):

   ```
   var f = function () {return this};
   f.call (4)          // => 4
   f.call (0)          // => 0
   f.call (false)      // => false
   f.call (null)       // => [object global]
   ```

[6] God forbid.

Given this unpredictability, most Javascript libraries provide a facility to set a function's `this` binding (referred to within Javascript circles as just a function's binding) to something invocation-invariant. The easiest way to do this is to define a function that proxies arguments using `apply` and closes over the proper value (luckily, closure variables behave normally):

```
var bind = function (f, this_value) {
  return function () {return f.apply (this_value, arguments)};
};
```

The difference between `call` and `apply` is straightforward: `f.call (x, y, z)` is the same as `f.apply (x, [y, z])`, which is the same as `bind (f, x) (y, z)`. That is, the first argument to both `call` and `apply` becomes `this` inside the function, and the rest are passed through. In the case of `apply` the arguments are expected in an array-like thing (`arguments` works here), and in the case of `call` they're passed in as given.

3.3.1 Important consequence: eta-reduction

In most functional programming languages, you can eta-reduce things; that is, if you have a function of the form `function (x) {return f (x)}`, you can just use `f` instead. But in Javascript that's not always a safe transformation; consider this code:

```
Array.prototype.each = function (f) {
  for (var i = 0, l = this.length; i < l; ++i)
    f (this[i]);
};

var xs = [];
some_array.each (function (x) {xs.push (x)});
```

It might be tempting to rewrite it more concisely as:

```
some_array.each (xs.push);
```

however this latter form will result in a mysterious Javascript error when the native `Array.push` function finds `this` to be the global object instead of `xs`. The reason should be apparent: when the function is called inside `each`, it is invoked as a function instead of a method. The fact that the function started out as a method on `xs` is forgotten. (Just like case 3 above.)

The simplest way around this is to bind `xs.push` to `xs`:

```
some_array.each (bind (xs.push, xs));
```

3.3.2 Odd tidbit: `this` is never falsy

For reasons explained in section 4.6, `this` will never be set to a falsy value. If you try to set it to `null` or `undefined`, say by doing this:

```
var f = function () {
  return this;
};
f.call (null);    // returns null, right?
```

it will in fact become the global `this`, usually `window` in the browser. If you use a falsy primitive, `this` will refer to a boxed version of that primitive. This has some counterintuitive consequences, covered in more detail in section 5.4.

4 Gotchas

Javascript is an awesome language just like Perl is an awesome language and Linux is an awesome operating system. If you know how to use it properly, it will solve all of your problems trivially (well, almost), and if you miss one of its subtleties you'll spend hours hunting down bugs. I've collected the things I've run into here, which should cover most of Javascript's linguistic pathology.[7]

4.1 Semicolon inference

You won't run into any trouble if you always end lines with semicolons. However, most browsers consider it to be optional, and there is one potential surprise lurking if you choose to omit them.

Most of the time Javascript does what you mean. The only case where it might not is when you start a line with an open-paren, like this:

```
var x = f
(y = x) (5)
```

Javascript joins these two lines, forming:

```
var x = f (y = x) (5)
```

The only way around this that I know of is to put a semicolon on the end of the first line.

[7] There is plenty of this pathology despite Javascript being generally an excellent language. This makes it ideal both for people who want to get things done, and for bug-connoisseurs such as myself.

4.2 Void functions

Every function returns a value. If you don't use a `return` statement, then your function returns `undefined`; otherwise it returns whatever you tell it to. This can be a common source of errors for people used to Ruby or Lisp; for instance,

```
var x = (function (y) {y + 1}) (5);
```

results in `x` being `undefined`. If you're likely to make this slip, there's an Emacs mode called "js2-mode" that identifies functions with no side-effects or return values, and it will catch most of these errors.[8]

4.3 `var`

Be careful how you define a variable. If you leave off the `var` keyword, your variable will be defined in the global scope, which can cause some very subtle bugs:

```
var f = function () {         // f is toplevel, so global
  var x = 5;                  // x is local to f
  y = 6;                      // y is global
};
```

As far as I know, the same is true in both types of `for` loop:

```
for (i = 0; i < 10; ++i)      // i is global
for (var i = 0; i < 10; ++i)  // i is local to the function
for (k in some_object)        // k is global
for (var k in some_object)    // k is local to the function
```

4.4 Lazy scoping and mutability

This is a beautiful disaster. Check this out:

```
var x = [];
for (var i = 0; i < 3; ++i)
  x[i] = function () { return i; };

x[0]();  // What will these be?
x[1]();
x[2]();
```

[8] Of course, that's if you're an Emacs person. If you prefer a *real* editor (wink), I wrote a custom JS highlighter that handles some cases better than the builtin one: http://github.com/spencertipping/js-vim-highlighter.

What will our three functions return when they are eventually called? You might expect them to return 0, 1, and 2, respectively, since those were the values of `i` when they were created. However they will actually all return 3. This is because of Javascript's lazy scoping: Upon creation, each function receives only a variable name and a scope in which to search for it; the value itself is not resolved until the time of invocation, at which point `i` will equal 3.

The simplest way to fix this is to wrap our assignment in an anonymous function that is evaluated immediately, introducing another layer of scope. The following code works because within the enclosing anonymous function, the value of `new_i` never changes.

```
for (var i = 0; i < 3; ++i)
  (function (new_i) {
    x[new_i] = function () { return new_i; };
  })(i);
```

By the way, you might be tempted to do this:

```
for (var i = 0; i < 3; ++i) {
  var j = i;
  x[j] = function () { return j; };
}
```

This won't work for same reason that our original example failed: `j` will be scoped to the nearest enclosing function (remember that Javascript's scoping is function level, not block level!), so its value is changing just as frequently as `i`'s.

4.5 Equality

Because the == is lame, these are all true in Javascript:

```
null  == undefined
null  == 0
false == ''
''    == 0
true  == 1
true  == '1'
'1'   == 1
```

So, *never use the == operator unless you really want this behavior*. Instead, use === (whose complement is `!==`), which behaves sensibly. In particular, === requires both operands to not only be the same-ish, but also be of the same type. It does referential comparison for boxed values and structural comparison for unboxed values. If one side is boxed and the other is unboxed, === will always return false. Because string literals are unboxed, though, you can use it there: `'foo' === 'fo' + 'o'`.

There is one case in particular where == is more useful than ===. If you want to find out whether something has a property table (i.e. isn't `null` or `undefined`), the easiest way to go about it is `(x == null)` rather than the more explicit `(x === null || x === undefined)`. Apart from this I can't imagine using == very often.[9]

> **Tidbit of pathology:** It turns out that == isn't even stable under truthiness. If `x = 0` and `y = new Number(0)`, then `x == y`, `!!x` is `false`, and `!!y` is `true`. Section 4.6 talks more about why this kind of thing happens.

4.6 Boxed vs. unboxed

Boxed values are always truthy and can store properties. Unboxed values will silently fail to store them; for example:[10]

```
var x = 5;
x.foo = 'bar';
x.foo     // => undefined; x is an unboxed number.

var x = new Number (5);
x.foo = 'bar';
x.foo     // => 'bar'; x is a pointer.
```

How does a sometimes-boxed value acquire a box? When you do one of these things:

1. Call its constructor directly, as we did above
2. Set a member of its prototype and refer to `this` inside that method (see section 5)
3. Pass it as the first argument to a function's `call` or `apply` method (see section 3.3.2)

All HTML objects, whether or not they're somehow native, will be boxed.

4.7 Things that will silently fail or misbehave

Javascript is very lenient about what you can get away with. In particular, the following are all perfectly legal:

```
[1, 2, 3].foo     // => undefined
[1, 2, 3][4]      // => undefined
1 / 0             // => Infinity
0 * 'foo'         // => NaN
```

[9] And, in fact, there are good security reasons not to do so; see section 4.8 for all the gory details.
[10] There are other consequences of boxing; see sections 4.11 and 4.12 for some examples.

This can be very useful. A couple of common idioms are things like these:

```
e.nodeType || (e = document.getElementById (e));
options.foo = options.foo || 5;
```

Also, the language will convert *anything* to a string or number if you use +. All of these expressions are strings:

```
null + [1, 2]             // => 'null1,2'
undefined + [1, 2]        // => 'undefined1,2'
3 + {}                    // => '3[object Object]'
'' + true                 // => 'true'
```

And all of these are numbers:

```
undefined + undefined     // => NaN
undefined + null          // => NaN
null + null               // => 0
{} + {}                   // => NaN
true + true               // => 2
0 + true                  // => 1
```

And some of my favorites:

```
null * false + (true * false) + (true * true)     // => 1
true << true << true                              // => 4
true / null                                       // => Infinity
```

4.8 Numeric coercion

This one caught me off guard recently. Javascript's type coercions sometimes have inconsistent properties. For example:

```
{}                  // truthy
!!{}                // coerce to boolean, truthy
+{}                 // coerce to number, NaN, which is falsy
[]                  // truthy
!![]                // coerce to boolean, truthy
+[]                 // coerce to number, 0, which is falsy

[] == false         // true (because [] is really zero, or something)
[] == 0             // true
[] == ''            // true (because 0 == '')
[] == []            // false (different references, no coercion)
[1] == [1]          // false (different references, no coercion)
[1] == +[1]         // true (right-hand side is number, coercion)
```

You need to watch out for things like this when you're using certain operators with non-numeric things. For example, this function will not tell you whether an array contains any truthy values:

```
var has_truthy_stuff = function (xs) {
  var result = 0;
  for (var i = 0, l = xs.length; i < l; ++i)
    result |= xs[i];
  return !!result;
};
has_truthy_stuff([{}, {}, 0])       // returns false
```

The reason `has_truthy_stuff` returns false is because when `{}` is coerced to a number, it becomes `NaN`, which is falsy in Javascript. Using `|=` with `NaN` is just like using it with `0`; nothing happens. So `result` remains `0` for all values of the array, and the function fails.

By the way, you can change what numeric coercion does by (re)defining the `valueOf` method:

```
+{valueOf: function () {return 42}}         // -> 42
Object.prototype.valueOf = function () {
  return 15;
};
Array.prototype.valueOf = function () {
  return 91;
};
+{}                 // -> 15
+[]                 // -> 91
+[1]                // -> 91
```

It's worth thinking about this a little bit because it has some interesting implications. First, `valueOf()` may not halt. For example:

```
Object.prototype.valueOf = function () {
  while (true);
};
{} == 5             // never returns; {} is coerced to a number
+{}                 // never returns
!{}                 // returns false; this bypasses valueOf()
```

Second, `valueOf` is just a regular Javascript function, so it can create security holes. In particular, suppose you're using `eval()` as a JSON parser (not a good idea, by the way) and didn't check the input for well-formedness first. If someone sends you `{valueOf: function () {while (true);}}`, then your app will hang the first time it coerces the object to a number (and this coercion can be implicit, like the `== 5` case above).

Tidbit of pathology: The numeric value of an array depends on its contents:

```
+[0]              // 0
+[1]              // 1
+[2]              // 2
+[[1]]            // 1
+[[[[[[[1]]]]]]]  // 1
+[1, 2]           // NaN
+[true]           // NaN
+['4']            // 4
+['0xff']         // 255
+['  0xff']       // 255
-[]               // 0
-[1]              // -1
-[1, 2]           // NaN
```

The built-in numeric coercion on arrays will fail with a stack overflow error if your array is deeply nested enough. For example:

```
for (var x = [], a = x, tmp, i = 0; i < 100000; ++i) {
  a.push(tmp = []);
  a = tmp;
}
a.push(42);       // the value we want, 100000 levels deep
x == 5            // stack overflow in V8
```

Fortunately, at least in V8, numeric coercion is still well-defined when you have an array that contains itself; so this example isn't nearly as much fun as it could be:[11]

```
var a = [];
a.push(a);
+a                // 0
```

4.9 Things that will loudly fail

There is a point where Javascript will complain. If you call a non-function, ask for a property of `null` or `undefined`, or refer to a global variable that doesn't exist,[12] then Javascript will throw a `TypeError` or `ReferenceError`. By extension, referring to local variables that don't exist causes a `ReferenceError`, since Javascript thinks you're talking about a global variable.

[11] Though this could easily change if you redefine `valueOf()`.

[12] To get around the error for this case, you can say `typeof foo`, where `foo` is the potentially nonexistent global. It will return `'undefined'` if `foo` hasn't been defined (or contains the value `undefined`).

4.10 Throwing things

You can `throw` a lot of different things, including unboxed values. This can have some advantages; in this code for instance:

```
try {
  ...
  throw 3;
} catch (n) {
  // n has no stack trace!
}
```

the `throw`/`catch` doesn't compute a stack trace, making exception processing quite a bit faster than usual. But for debugging, it's much better to throw a proper error:

```
try {
  ...
  throw new Error(3);
} catch (e) {
  // e has a stack trace, useful in Firebug among other things
}
```

4.11 Be careful with `typeof`

Because it behaves like this:

```
typeof function () {}     // => 'function'
typeof [1, 2, 3]          // => 'object'
typeof {}                 // => 'object'
typeof null               // => 'object'
typeof typeof             // hangs forever in Firefox
```

`typeof` is a really lame way to detect the type of something in many cases.[13] Better is to use an object's `constructor` property, like this:

```
(function () {}).constructor      // => Function
[1, 2, 3].constructor             // => Array
({}).constructor                  // => Object
true.constructor                  // => Boolean
null.constructor                  // TypeError: null has no properties
```

In order to defend against `null` and `undefined` (neither of which let you ask for their constructor), you might try to rely on the falsity of these values:

```
x && x.constructor
```

[13] And because it returns a string, it's marginally slower than using `.constructor`.

But in fact that will fail for `''`, `0`, `false`, `NaN`, and possibly others. The only way I know to get around this is to just do the comparison:

```
x === null || x === undefined ? x : x.constructor
x == null ? x : x.constructor       // same thing, but more concise
```

Alternatively, if you just want to find out whether something is of a given type, you can just use `instanceof`, which never throws an exception.[14]

4.12 Also be careful with `instanceof`

`instanceof` is generally more useful than `typeof`, but it only works with boxed values. For example, these are all false:

```
3 instanceof Number
'foo' instanceof String
true instanceof Boolean
```

However, these are all true:

```
[] instanceof Array
({}) instanceof Object
[] instanceof Object        // Array inherits from Object
/foo/ instanceof RegExp     // regular expressions are always boxed
(function () {}) instanceof Function
```

One way to work around the first problem is to wrap primitives:

```
new Number(3) instanceof Number             // true
new String('foo') instanceof String         // also true
new Boolean(true) instanceof Boolean        // also true
```

In general, `(new x.constructor(x) instanceof x.constructor)` will be true for all primitive `x`. However, this doesn't hold for `null` or `undefined`. These will throw errors if you ask for their constructors, and as far as I know are never returned from the result of a constructor invocation (using `new`, that is).

4.13 Browser incompatibilities

Generally browsers since IE6 have good compatibility for core language stuff. One notable exception, however, is an IE bug that affects `String.split`:

```
var xs = 'foo bar bif'.split (/(\s+)/);
xs          // on reasonable browsers: ['foo', ' ', 'bar', ' ', 'bif']
xs          // on IE: ['foo', 'bar', 'bif']
```

[14] Well, almost. If you ask for it by putting `null`, `undefined`, or similarly inappropriate things on the right-hand side you'll get a `TypeError`.

A more subtle bug that took me several hours to find is that IE6 also doesn't return functions from `eval()`:

```
var f = eval('function() {return 5}');
f()         // on reasonable browsers: 5
f()         // on IE6: 'Object expected' (because f is undefined)
```

I'm sure there are other similar bugs out there, though the most common ones to cause problems are generally in the DOM.[15]

5 Prototypes

I used to have a very anti-OOP comment here, but considering that I occasionally use prototypes I removed it. Despite my obvious and probably unfair vendetta against Javascript's linguistic compromises to pander to Java-inspired marketing pressure,[16] prototype-based programming can be useful on occasion. This section contains my subjective and biased view of it.

Whenever you define a function, it serves two purposes. It can be what every normal programmer assumes a function is – that is, it can take values and return values, or it can be a mutant instance-generating thing that does something completely different. Here's an example:

```
// A normal function:
var f = function (x) {return x + 1};
f (5)     // => 6
```

This is what most people expect. Here's the mutant behavior that no rational person would ever imagine:

```
// A constructor function
var f = function (x) {this.x = x + 1};    // no return!
var i = new f (5);                        // i.x = 6
```

The following things are true at this point:

```
i.constructor === f
i.__proto__ === i.constructor.prototype   // on Firefox, anyway
i instanceof f
typeof i === 'object'
```

The `new` keyword is just a right-associative (prefix) unary operator, so you can instantiate things first-class:

[15] jQuery is your friend here. It's branded as a Javascript library, but in fact it's a set of enhancements to the DOM to (1) achieve a uniform cross-browser API, and (2) make it easier to retrieve and manipulate nodes.

[16] Hence its name, *Java*script, despite all of the dissimilarities.

```
var x = 5;
new x.constructor ();     // Creates a boxed version of x, regardless of what x is
new new Function('x', 'this.x = 5');
```

If you are going to program using this questionable design pattern, then you'll probably want to add methods to things:[17]

```
var f = function (x) {this.x = x};
f.prototype.add_one = function () {++this.x};
var i = new f (5);
i.add_one ();
i.x               // => 6
```

You can find tons of information about this kind of prototype programming online.

5.1 Why `new` is awful

`new` has some cool features (such as being first-class), but it has a really horrible shortcoming. Most functions in Javascript can be *forwarded* – that is, you can write a new function to wrap an existing one and the function being called will never know the difference. For example:

```
var to_be_wrapped = function (x) {return x + 1};
var wrapper       = function () {
  return to_be_wrapped.apply (this, arguments);
};
// for all x, wrapper(x) === to_be_wrapped(x)
```

However, `new` has no such mechanism. You can't forward a constructor in the general case, because `new` has no equivalent of `apply`. (Though this isn't the whole story; see the next section for a brilliant workaround.)

5.2 Why `new` isn't quite so awful

I recently received an e-mail from Ondrej Zara explaining that my bias against `new` was ill-founded, and containing a remarkably elegant workaround for the problem I complained about in the previous section. Here's his implementation verbatim:

```
var Forward = function(ctor /*, args... */) {
  var tmp = function(){};
  tmp.prototype = ctor.prototype;
  var inst = new tmp();
  var args = [];
```

[17] This section used to say that `i.x` would evaluate to 7. That isn't true though. It's actually 6, as indicated. (Thanks to Daniel Gasparotto for pointing this out.)

```
  for (var i=1;i<arguments.length;i++) { args.push(arguments[i]); }
  ctor.apply(inst, args);
  return inst;
}
```

And the use case:

```
var Class = function(a, b, c) {}
var instance = Forward(Class, a, b, c);
instance instanceof Class; // true
```

At first I was very skeptical that this approach would work, but I have yet to find a case where it fails. So constructors can indeed be forwarded in Javascript, despite my previous claims to the contrary.

5.3 Why you should use prototypes

If you need a dynamic-dispatch pattern, then prototypes are probably your best bet and you should use them rather than a roll-your-own approach. Google's V8 has a bunch of prototype-specific optimizations, as do later releases of Firefox. Also, prototypes save memory; having a pointer to a prototype is much cheaper than having n pointers to n attributes.

If, on the other hand, you find yourself implementing actual inheritance hierarchies, then you're probably making a mistake.[18] I have found prototypes to be an effective way to program in Javascript, but inheritance in Javascript is (1) slow,[19] and (2) poorly representative of Javascript's "everything is public" model.

5.4 Autoboxing

You might be tempted to try something like this:[20]

```
Boolean.prototype.xor = function (rhs) {return !! this !== !! rhs};
```

And, upon running this code, you'd run into this tragically unfortunate property:

```
false.xor (false)         // => true
```

[18] OK, I'm being biased about this point. I tend to treat Javascript more like Scheme than like Smalltalk, so I don't think much in terms of classical object-oriented modeling. Also, since closures are really fast, it's OK to use functional abstraction instead of inheritance. Javascript tends to be better suited to metaprogramming than inheritance.

[19] In some cases really slow. The difference between single-level and multiple-level prototype lookups in Firefox 3.5, for instance, is enormous.

[20] `!! x` is just an idiom to make sure that `x` ends up being a boolean. It's a double-negation, and `!` always returns either `true` or `false`.

The reason is that when you treat an unboxed value as an object (e.g. invoke one of its methods), it gets temporarily promoted into a boxed value for the purposes of that method call. This doesn't change its value later, but it does mean that it loses whatever falsity it once had. Depending on the type you're working with, you can convert it back to an unboxed value:

```
function (rhs) {return !! this.valueOf () !== !! rhs};
```

6 A Really Awesome Equality

There is something really important about Javascript that isn't at all obvious from the way it's used. It is this: The syntax `foo.bar` is, in all situations, identical to `foo['bar']`. You could safely make this transformation to your code ahead of time, whether on value-properties, methods, or anything else. By extension, you can assign non-identifier things to object properties:

```
var foo = [1, 2, 3];
foo['@snorkel!'] = 4;
foo['@snorkel!']    // => 4
```

You can also read properties this way, of course:

```
[1, 2, 3]['length']         // => 3
[1, 2, 3]['push']           // => [native function]
```

In fact, this is what the `for (var ... in ...)` syntax was built to do: Enumerate the properties of an object. So, for example:

```
var properties = [];
for (var k in document) properties.push (k);
properties          // => a boatload of strings
```

However, `for ... in` has a dark side. It will do some very weird things when you start modifying prototypes. For example:

```
Object.prototype.foo = 'bar';
var properties = [];
for (var k in {}) properties.push (k);
properties          // => ['foo']
```

To get around this, you should do two things. First, never modify `Object`'s prototype, since everything is an instance of `Object` (including arrays and all other boxed things); and second, use `hasOwnProperty`:[21]

[21] OK, so you're probably wondering why we don't see the `hasOwnProperty` method from a `for ... in` loop, since it's obviously a property. The reason is that Javascript's attributes have invisible flags (as defined by the ECMAScript standard), one of which is called `DontEnum`. If `DontEnum` is set for some attribute, then a `for ... in` loop will not enumerate it. Javascript doesn't provide a way to set the `DontEnum` flag on anything you add to a prototype, so using `hasOwnProperty` is a good way to prevent looping over other people's prototype extensions. Note that it fails sometimes on IE6; I believe it always returns false if the prototype supplies an attribute of the same name.

```
Object.prototype.foo = 'bar';
var properties = [], obj = {};
for (var k in obj) obj.hasOwnProperty (k) && properties.push (k);
properties          // => []
```

And very importantly, never use `for ... in` to iterate through arrays (it returns string indices, not numbers, which can cause problems) or strings. Either of these will fail if you add methods to `Array` or `String` (or `Object`, but you shouldn't do that).

7 If You Have 20 Minutes...

Javascript can do almost anything that other languages can do. However, it might not be very obvious how to go about it.

7.1 Iterators for cool people

Because languages like Ruby showed the world just how passé `for` loops really are, a lot of self-respecting functional programmers don't like to use them. If you're on Firefox, you won't have to; the `Array` prototype includes `map` and `forEach` functions already. But if you're writing cross-browser code and aren't using a library that provides them for you, here is a good way to implement them:

```
Array.prototype.each = Array.prototype.forEach || function (f) {
  for (var i = 0, l = this.length; i < l; ++i)
    f (this[i]);
  return this;      // convenient for chaining
};

Array.prototype.map = Array.prototype.map || function (f) {
  var ys = [];
  for (var i = 0, l = this.length; i < l; ++i)
    ys.push (f (this[i]));
  return ys;
};
```

As far as I know this is (almost) the fastest way to write these functions. We declare two variables up-front (`i` and `l`) so that the length is cached; Javascript won't know that `this.length` is invariant with the `for` loop, so it will check it every time if we fail to cache it. This is expensive because due to boxing we'd have a failed hash-lookup on `this` that then dropped down to `this.__proto__`, where it would find the special property `length`. Then, a method call would happen to retrieve `length`.[22]

[22]This gets into how Javascript presents certain APIs. Internally it has a notion of gettable and settable properties, though there isn't a cross-browser way to create them. But properties such as

The only further optimization that could be made is to go through the array backwards (which only works for `each`, since `map` is assumed to preserve order):

```
Array.prototype.each = function (f) {
  for (var i = this.length - 1; i >= 0; --i)
    f (this[i]);
};
```

This ends up being very slightly faster than the first implementation because it changes a floating-point subtraction (required to evaluate < for non-zero quantities) into a sign check, which internally is a bitwise `and` and a zero-predicated jump. Unless your Javascript engine inlines functions and you're really determined to have killer performance (at which point I would ask why you're using Javascript in the first place), you probably never need to consider the relative overhead of a non-zero < vs. a zero >=.

You can also define an iterator for objects, but not like this:

```
// NO NO NO!!! Don't do it this way!
Object.prototype.each = function (f) {
  for (var k in this) this.hasOwnProperty (k) && f (k);
};
```

Much better is to implement a separate `keys` function to avoid polluting the `Object` prototype:

```
var keys = function (o) {
  var xs = [];
  for (var k in o) o.hasOwnProperty (k) && xs.push (k);
  return xs;
};
```

7.2 Java classes and interfaces

No sane person would ever want to use these. But if you're insane or are being forced to, then the Google Web Toolkit will give you a way to shoot yourself in the foot and turn it into Javascript.

7.3 Recursive metaclasses

There are different ways to approach this, but a straightforward way is to do something like this:[23]

`length`, `childNodes`, etc. are all really method calls and not field lookups. (Try assigning to one and you'll see.)

[23] Remember that a class is just a function that produces instances. Nothing about the `new` keyword is necessary to write object-oriented code (thank goodness).

```
var metaclass = {methods: {
  add_to: function (o) {
    var t = this;
    keys (this.methods).each (function (k) {
      o[k] = bind (t.methods[k], o);        // can't use /this/ here
    });
    return o}};
metaclass.methods.add_to.call (metaclass, metaclass);
```

At this point, `metaclass` is now itself a `metaclass`. We can start to implement instances of it:

```
var regular_class = metaclass.add_to ({methods: {}});
regular_class.methods.def = function (name, value) {
  this.methods[name] = value;
  return this;
};
regular_class.methods.init = function (o) {
  var instance = o || {methods: {}};
  this.methods.init && this.methods.init.call (instance);
  return this.add_to (instance);
};
regular_class.add_to (regular_class);
```

This is a Ruby-style class where you can define public methods and a constructor. So, for example:

```
var point = regular_class.init ();
point.def ('init', function () {this.x = this.y = 0});
point.def ('distance', function () {
  return Math.sqrt (this.x * this.x + this.y * this.y)});
```

We're using the rather verbose `this.x`, which may offend some Python-eschewing Rubyists. Fortunately, we can use dynamic rewriting to use the `$` where Rubyists would use `@`:[24]

```
var ruby = function (f) {
  return eval (f.toString ().replace (/\$(\w+)/g,
    function (_, name) {return 'this.' + name}));
};

point.def ('init', ruby (function () {$x = $y = 0}));
point.def ('distance', ruby (function () {
  return Math.sqrt ($x * $x + $y * $y)}));
```

[24] And, in fact, we could bake this `ruby()` transformation into a metaclass to make it totally transparent if we wanted to.

And now you can use that class:

```
var p = point.init ();
p.x = 3, p.y = 4;
p.distance ()       // => 5
```

The advantage of using metaclasses is that you can do fun stuff with their structure. For example, suppose that we want to insert method tracing into all of our points for debugging purposes:[25]

```
keys (point.methods).each (function (k) {
  var original = point.methods[k];
  point.methods[k] = function () {
    trace ('Calling method ' + k + ' with arguments ' +
           Array.prototype.join.call (arguments, ', '));
    return original.apply (this, arguments);
  };
});
```

Now `trace` (which isn't a Javascript built-in, so you'd have to define it) would be called each time any method of a `point` instance was called, and it would have access to both the arguments and the state.

7.4 Tail calls

Javascript does not do tail-call optimization by default, which is a shame because some browsers have short call stacks (the shortest I'm aware of is 500 frames, which goes by especially quickly when you have bound functions and iterators). Luckily, encoding tail calls in Javascript is actually really simple:

```
Function.prototype.tail = function () {return [this, arguments]};
Function.prototype.call_with_tco = function () {
  var c      = [this, arguments];
  var escape = arguments[arguments.length - 1];
  while (c[0] !== escape)
    c = c[0].apply (this, c[1]);
  return escape.apply (this, c[1]);
};
```

We can now use this definition to write a tail-call optimized factorial function:[26]

[25] The example here used to contain the expression `arguments.join`, which is invalid – `arguments` isn't an array. Now it uses the "pretend this is an array for the purposes of calling `join` on it" idiom, which usually works. (Though you'll sometimes get errors about methods not being generalized, as is the case on Chrome if you try to use `Array.prototype.toString()` this way.)

[26] This technique is called trampolining and doesn't constitute implementing delimited continuations, as I found out later. However, it's still pretty cool.

```
// Standard recursive definition
var fact1 = function (n) {
  return n > 0 ? n * fact1 (n - 1) : 1;
};

// Tail-recursive definition
var fact2 = function (n, acc) {
  return n > 0 ? fact2 (n - 1, acc * n) : acc;
};

// With our tail-call mechanism
var fact3 = function (n, acc, k) {
  return n > 0 ? fact3.tail (n - 1, acc * n, k) : k.tail (acc);
};
```

The first two functions can be called normally:

```
fact1 (5)           // => 120
fact2 (5, 1)        // => 120
```

though neither will run in constant stack space. The third one, on the other hand, will if we call it this way:

```
var id = function (x) {return x};
fact3.call_with_tco (5, 1, id)      // => 120
```

The way this tail-call optimization strategy works is that instead of creating new stack frames:

```
fact1(5)
  5 * fact1(4)
    4 * fact1(3)
    ...
```

or even creating hollow ones:

```
fact2(5, 1)
  fact2(4, 5)
    fact2(3, 20)
    ...
```

we pop out of the last stack frame before allocating a new one (treating the array of `[function, args]` as a kind of continuation to be returned):

```
fact3(5, 1, k) -> [fact3, [4, 5, k]]
fact3(4, 5, k) -> [fact3, [3, 20, k]]
fact3(3, 20, k) ...
```

It isn't a bad performance hit, either – the overhead of allocating a two-element array of pointers is minimal.

7.5 Syntactic macros and operator overloading

Lazy scoping lets us do some cool stuff. Let's say we want to define a new syntax form for variable declaration, so that instead of this:

```
var f = function () {
  var y = (function (x) {return x + 1}) (5);
  ...
};
```

we could write this:

```
var f = function () {
  var y = (x + 1).where (x = 5);
  ...
};
```

This can be implemented in terms of regular expressions if we don't mind being woefully incorrect about half the time:

```
var expand_where = function (f) {
  var s = f.toString ();
  return eval (s.replace (/\(([^)]+)\)\.where\(([^)])\)/,
    function (_, body, value) {
      return '(function (' + value.split ('=')[0] + '){return ' +
             body + '}) (' + value.split ('=', 2)[1] + ')';
    }));
};
```

Now we can say this:

```
var f = expand_where (function () {
  var y = (x + 1).where (x = 5);
  ...
});
```

Obviously a proper parser is more appropriate because it wouldn't fail on simple paren boundaries. But the important thing is to realize that a function gives you a way to quote code, just like in Lisp:

```
(defmacro foo (bar) ...)
(foo some-expression)
```

becomes this in Javascript (assuming the existence of `parse` and `deparse`, which are rather complicated):[27]

[27] Real versions of these are implemented in http://github.com/spencertipping/caterwaul, if you're interested to see what they look like. It's also a reasonable reference for syntactic edge cases.

```
var defmacro = function (transform) {
  return function (f) {
    return eval (deparse (transform (parse (f.toString ()))));
  };
};
var foo = defmacro (function (parse_tree) {
  return ...;
});
foo (function () {some-expression});
```

This principle can be extended to allow for operator overloading if we write a transformation that rewrites operators into method calls:

```
x << y      // becomes x['<<'](y)
```

Remember that property names aren't restricted to identifiers – so we could overload the << operator for arrays to work like it does in Ruby with:

```
Array.prototype['<<'] = function () {
  for (var i = 0, l = arguments.length; i < l; ++i)
    this.push (arguments[i]);
  return this;
};
```

The only thing that's unfortunate about implementing this stuff in Javascript rather than Lisp is that Javascript bakes syntactic constructs into the grammar, so trying to introduce new syntactic forms such as `when` isn't very convenient:

```
expand_when (function () {
  when (foo) {      // compile error; { unexpected
    bar ();
  }
});
```

But anything you can do inside the Javascript parse tree is fair game.[28]

8 Further reading

I highly recommend reading jQuery (http://jquery.com) for the quality and conscientiousness of the codebase. It's a brilliant piece of work and I've learned a tremendous amount by pawing around through it.

Douglas Crockford has written some excellent Javascript references, including the well-known *Javascript: The Good Parts* and a less-well-known but free

[28] Keep in mind that `toString` will sometimes rewrite your function to standard form, so leveraging ambiguities of the syntax isn't helpful. In Firefox, for example, writing expressions with excess parentheses is not useful because those excess parentheses are lost when you call `toString`.

online tour of the language at http://javascript.crockford.com/survey.html.[29]

As a shameless plug, I also recommend reading through Divergence (http://github.com/spencertipping/divergence), a library that I wrote. It's very different from jQuery – much more terse and algorithmic (and has no DOM involvement). jQuery uses a more traditional approach, whereas Divergence tends to make heavy use of closures and functional metaprogramming.

If you're into Lisp and metaprogramming, you might also enjoy http://github.com/spencertipping/divergence.rebase and http://github.com/spencertipping/caterwaul, two projects that use function serialization and `eval()` to implement some of the syntactic extensions mentioned in the last section.

Also, I recently found a site called http://wtfjs.com that seems to be dedicated to exposing all of Javascript's edge-case pathologies. It's quite a fun and enlightening read. A more in-depth look at the good, bad, and ugly parts of Javascript is http://perfectionkills.com; this site is written by one of the PrototypeJS developers and has convinced me that I really don't know Javascript that well.

[29] There are some discrepancies between his view of Javascript and mine. Neither is incorrect, there are just different unstated assumptions. For example, when he says that there are three primitives he is correct; he counts types by the number of unboxed representations, whereas I count them by the number of literal constructors.