

Intermediate Javascript

In this lecture we'll go through the history of JS, talk about the important global objects and built-ins that are common across both client- and server-side JS implementations, and discuss the functional programming (FP) and object-oriented (OOP) paradigms in the context of JS.

The rise of Javascript

Javascript (JS) began as an add-on to Netscape, programmed in a few days by Brendan Eich in the early 1990s ([history](#), [more](#), [more](#)). Though JS is a completely different language from Java, Netscape used a similar syntax to Java and named it “Javascript” to piggy-back on Java’s marketing dollars:

Brendan Eich: The big debate inside Netscape therefore became “why two languages? why not just Java?” The answer was that two languages were required to serve the two mostly-disjoint audiences in the programming ziggurat who most deserved dedicated programming languages: the component authors, who wrote in C++ or (we hoped) Java; and the “scripters”, amateur or pro, who would write code directly embedded in HTML. Whether any existing language could be used, instead of inventing a new one, was also not something I decided. The diktat from upper engineering management was that the language must “look like Java”.

You can construct *some* rationale for this at the time ([Java applets](#) would be used for heavyweight web apps, with JS for lightweight page modifications), but in general the nomenclature issue mainly causes confusion. JS is not a stripped down version of Java, it is a completely different language.

And for much of its early history JS was not respected as a *serious* language. JS was seen as a way to annoy visitors with popup alerts and annoying animations. All that changed with the launch of Gmail in 2004. An influential [reverse-engineering](#) of Gmail and Google Maps by Jesse James Garrett in 2005 spawned the term AJAX, and it became clear that JS apps could actually accomplish some wondrous things. Even during the AJAX era, JS was still a client side language through the 2000s, with interpreters of the language mostly limited to the browser environment. In the mid-2000s, projects like [Mozilla Rhino](#) arose to provide a server-side JS environment, but it never caught on like [CPython](#) or [CRuby](#) (the most popular interpreters for the abstract Python and Ruby languages¹ respectively).

¹Though people often refer to them interchangeably, there is a difference between the Javascript programming language (an abstract entity specified by the ECMAScript [language specification](#)) and a particular Javascript interpreter (like Chrome’s embedded v8 interpreter, Mozilla Rhino, or node.js). Think of the language specification itself as giving a lowest common denominator that all interpreters must be able to parse and execute. On top of this, each vendor may then customize their interpreter to add JS extensions or new default libraries, above and beyond what is in the current specification. Often these extensions in turn become incorporated in the next specification. In this manner we have the cycle of innovation, then consensus, then innovation, and so on.

However, in the late 2000s a breakthrough in the optimization of dynamic languages called [trace trees](#) spurred the development of extremely fast JS compilers by both Google Chrome ([the v8 engine](#)) and Mozilla Firefox ([Tracemonkey](#)). Then in late 2009, an enterprising young engineer named Ryan Dahl took Chrome's open source v8 code, factored it out of the browser environment, and [announced](#) a new server-side JS environment on top of v8 called node.js. Not only did node popularize server-side programming in JS, it made native use of JS's facilities for non-blocking IO ([read more](#)), and is starting to become quite a [popular](#) choice among new startups - including the increasingly high-profile [Medium.com](#), by the founder of Blogger and co-founder of Twitter.

Basics and Built-ins

Let's now illustrate some intermediate JS concepts. Most of these examples should work both in a node environment and in a browser's JS prompt², with the exception of the `require` invocations. We assume you know how to program and have reviewed this [short tutorial](#) on MDN, where variable types, control flow primitives, conditional expressions, the syntax of function definitions, and so on are reviewed. Perhaps the best way³ to confirm that you understand these in the context of JS is to do a few problems from [Project Euler](#). Here's an example with [Euler problem 1](#):

```
1 #!/usr/bin/env node
2 // http://projecteuler.net/problem=1
3 // If we list all the natural numbers below 10 that are multiples of 3 or 5,
4 // we get 3, 5, 6 and 9. The sum of these multiples is 23.
5 //
6 // Find the sum of all the multiples of 3 or 5 below 1000.
7
8 var divides = function(a, b) {
9     return b % a === 0;
10 };
11
12 var anydivide = function(as, b) {
13     for(var ii in as) {
14         if(divides(as[ii], b)) {
15             return true;
16         }
17     }
18     return false;
19 };
20
21 var sum = function(arr) {
22     var cc = 0;
```

²In Chrome, you can view the JS prompt by going to the View Menu item, selecting Developer, and then selecting the Javascript Console. The variables in the webpage you're currently viewing will then be accessible at the prompt.

³If you want to do something for extra credit, it would also probably be quite useful to fork the [PLEAC](#) project and add Javascript. PLEAC predates node, and so many of the server-side examples can and should be redone in node now that there is a popular SSJS interpreter.

```

23     for(var ii in arr) {
24         cc += arr[ii];
25     }
26     return cc;
27 };
28
29 var fizzbuzz = function(factors, max) {
30     var out = [];
31     for(var nn = 1; nn < max; nn += 1) {
32         if(anydivide(factors, nn)) {
33             out.push(nn);
34         }
35     }
36     return sum(out);
37 };
38
39 console.log(fizzbuzz([3, 5], 1000));

```

We use the name `fizzbuzz` for the name of the final function as a homage to the [original problem](#) from which the Project Euler problem is derived. Here's another example with [Euler problem 2](#):

```

1  #!/usr/bin/env node
2  // http://projecteuler.net/problem=2
3  //
4  // Each new term in the Fibonacci sequence is generated by adding the
5  // previous two terms. By starting with 1 and 2, the first 10 terms will be:
6  //
7  // 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
8  //
9  // By considering the terms in the Fibonacci sequence whose values do not
10 // exceed four million, find the sum of the even-valued terms.
11
12 // Fibonacci: closed form expression
13 // wikipedia.org/wiki/Golden_ratio#Relationship_to_Fibonacci_sequence
14 var fibonacci = function(n) {
15     var phi = (1 + Math.sqrt(5))/2;
16     return Math.round((Math.pow(phi, n + 1) - Math.pow(1 - phi, n + 1))/Math.sqrt(5));
17 };
18
19 var iseven = function(n) {
20     return n % 2 === 0;
21 };
22
23 var sum = function(arr) {
24     var cc = 0;
25     for(var ii in arr) {

```

```

26         cc += arr[ii];
27     }
28     return cc;
29 };
30
31 var fibsum = function(max) {
32     var value = 0;
33     var ii = 1;
34     var out = [];
35     var flag = false;
36     while(value < max) {
37         value = fibonacci(ii);
38         flag = iseven(value);
39         ii += 1;
40         if(flag && value < max) {
41             out.push(value);
42         }
43     }
44     return sum(out);
45 };
46
47 console.log(fibsum(4e6));

```

After a few of these [Project Euler](#) problems, you should feel comfortable with doing basic arithmetic, writing recursive functions, doing some string processing, and translating general CS algorithms into JS.

Array

Next, let's dive into some JS specifics, starting with the built-in [global objects](#) present in all full-fledged JS environments and specifically beginning with [Array](#). The [Array](#) global has some fairly self-evident methods that allow us to populate, shrink, expand, and concatenate lists of items.

```

1  // Important Array global methods
2
3  // length
4  var log = console.log;
5  var foo = [1, 2, 3, 3, 6, 2, 0];
6  log(foo.length); // 7
7
8  // concat: create new array
9  var bar = [4, 5, 6];
10 var baz = foo.concat(bar);
11
12 log(foo); // [ 1, 2, 3, 3, 6, 2, 0 ]
13 log(bar); // [ 4, 5, 6 ]

```

```

14 log(baz); // [ 1, 2, 3, 3, 6, 2, 0, 4, 5, 6 ]
15
16 // push/unshift to add to end/beginning
17 foo.push(10);
18 foo.unshift(99);
19 log(foo); // [ 99, 1, 2, 3, 3, 6, 2, 0, 10 ]
20
21 // pop/shift to remove the last/first element
22 var last = foo.pop();
23 var first = foo.shift();
24 log(last); // 10
25 log(first); // 99
26 log(foo); // [ 1, 2, 3, 3, 6, 2, 0 ]
27
28 // join: combine elements in an array into a string
29 log('<tr><td>' + bar.join('</td><td>') + "</td></tr>");
30 // <tr><td>4</td><td>5</td><td>6</td></tr>
31
32 // slice: pull out a subarray
33 var ref = foo.slice(3, 6);
34 log(ref); // [ 3, 6, 2 ]
35 log(foo); // [ 1, 2, 3, 3, 6, 2, 0 ]
36 ref[0] = 999;
37 log(ref); // [ 999, 6, 2 ]
38 log(foo); // [ 1, 2, 3, 3, 6, 2, 0 ]
39
40 // reverse: MODIFIES IN PLACE
41 foo.reverse();
42 log(foo); // [ 0, 2, 6, 3, 3, 2, 1 ]
43 foo.reverse();
44 log(foo); // [ 1, 2, 3, 3, 6, 2, 0 ]
45
46 // sort: MODIFIES IN PLACE
47 foo.sort();
48 log(foo); // [ 0, 1, 2, 2, 3, 3, 6 ]
49
50 // inhomogeneous types work
51 var arr = [99, 'mystr', {'asdf': 'alpha'}];
52
53 // JS will do the best it can if you invoke default methods.
54 // So make sure to use the REPL if you're doing something exotic.
55 arr.sort();
56 log(arr); // [ 99, { asdf: 'alpha' }, 'mystr' ]
57 arr.join(','); // '99,[object Object],mystr'
58
59
60 // Be very careful in particular when working with inhomogeneous arrays that

```

```

61 // contain objects. Methods like slice will not do deep copies, so modifications
62 // of the new array can propagate back to the old one when dealing with objects.
63 var newarr = arr.slice(0, 2);
64 console.log(newarr);      // [ 99, { asdf: 'alpha' } ]
65 console.log(arr);        // [ 99, { asdf: 'alpha' }, 'mystr' ]
66 newarr[1].asdf = 'ooga';
67 console.log(newarr);      // [ 99, { asdf: 'ooga' } ]
68 console.log(arr);        // [ 99, { asdf: 'ooga' }, 'mystr' ]

```

Importantly, as shown above, unlike an array in C, an Array in JS can be heterogenous in that it contain objects of different types. It's useful to use the REPL to verify the results if you're doing any type conversions with elements in arrays, making copies of arrays, or working with arrays that have nested objects in them.

Date

The [Date](#) object is useful for doing arithmetic with dates and times. While this might seem simple, you generally don't want to implement these kinds of routines yourself; date and time arithmetic involves tricky things like leap years and daylight savings time. A popular convention for dealing with this is to think of all times in terms of milliseconds relative⁴ to the [Unix Epoch](#), namely January 1 1970 at 00:00:00 GMT. Let's go through a few examples of working with Dates, building up to a concrete example of doing a JSON request and parsing string timestamps into Date instances.

```

1 // 1. Constructing Date instances.
2 //
3 //     Make sure to use "new Date", not "Date" when instantiating.
4 //
5 var year = 2000;
6 var month = 02;
7 var day = 29;
8 var hour = 12;
9 var minute = 30;
10 var second = 15;
11 var millisecond = 93;
12 var milliseconds_since_jan_1_1970 = 86400 * 10 * 1000;
13 var iso_timestamp = '2003-05-23T17:00:00Z';
14
15 var dt0 = Date(); // BAD - just a string, not a Date instance
16 var dt1 = new Date(); // GOOD - a real Date instance using new
17 var dt2 = new Date(milliseconds_since_jan_1_1970);
18 var dt3 = new Date(iso_timestamp);

```

⁴The problem with this convention is that after 2147483647 milliseconds, we arrive at January 19, 2038 3:14:07 GMT. If a 32-bit integer has been used for keeping track of the Unix time, it will overflow after this time and the system will crash (because $2^{31} = 2147483648$). The solution is to use a 64-bit integer for all time arithmetic going forward, but many legacy systems may not be updated in time. This is called the [Year 2038 problem](#), like the Year 2000 problem.

```

19 var dt4 = new Date(year, month, day);
20 var dt5 = new Date(year, month, day, hour, minute, second, millisecond);
21
22 /*
23 > dt1
24 Mon Aug 05 2013 07:19:45 GMT-0700 (PDT)
25 > dt2
26 Sat Jan 10 1970 16:00:00 GMT-0800 (PST)
27 > dt3
28 Fri May 23 2003 10:00:00 GMT-0700 (PDT)
29 > dt4
30 Wed Mar 29 2000 00:00:00 GMT-0800 (PDT)
31 > dt5
32 Wed Mar 29 2000 12:30:15 GMT-0800 (PDT)
33 */
34
35 // 2. Date classmethods - these return milliseconds, not Date instances.
36
37 // now: gives number of milliseconds since Jan 1, 1970 00:00 UTC
38 var milliseconds_per_year = (86400 * 1000 * 365);
39 var years_from_epoch = function(ms) {
40     console.log(Math.floor(ms/milliseconds_per_year));
41 };
42 years_from_epoch(Date.now()); // 43
43
44 // parse: takes in ISO timestamp and returns milliseconds since epoch
45 years_from_epoch(Date.parse('2003-05-23T17:00:00Z')); // 33
46
47 // UTC: Constructor that returns milliseconds since epoch
48 years_from_epoch(Date.UTC(year, month, day, hour, minute,
49                             second, millisecond)); // 30
50
51
52 // 3. Date Examples
53
54 // 3.1 - Calculating the difference between two dates
55 console.log(dt3); // Fri May 23 2003 10:00:00 GMT-0700 (PDT)
56 console.log(dt4); // Wed Mar 29 2000 00:00:00 GMT-0800 (PDT)
57 var ddt = dt3 - dt4;
58 var ddt2 = dt3.getTime() - dt4.getTime();
59 console.log(ddt); // 99392400000
60 console.log(ddt2); // 99392400000
61 console.log(ddt / milliseconds_per_year); // 3.151712328767123
62
63 // 3.2 - Get JSON data, parsing strings into Date instances,
64 //         and then return docs structure.
65 var data2docs = function(data) {

```

```

66     var docs = [];
67     var offset = 'T12:00:00-05:00'; // Explicitly specify time/timezone.
68     var dt, ii, dtnew;
69     for(ii = 0; ii < data.results.length; ii += 1) {
70         dt = data.results[ii].publication_date; // looks like: '2012-12-14'
71         dti = new Date(dt + offset);
72         docs.push({'title':data.results[ii].title, 'publication_date': dti});
73     }
74     return docs;
75 };
76
77 var docs2console = function(docs) {
78     for(var ii = 0; ii < docs.length; ii++) {
79         doc = docs[ii];
80         console.log('Date: %s\nTitle: %s\n', doc.publication_date, doc.title);
81     }
82 };
83
84 var apiurl = "https://www.federalregister.gov/api/v1/articles/" +
85 "03-12969,2012-30312,E8-24781.json?fields%5B%5D=title" +
86 "&fields%5B%5D=publication_date";
87 var request = require('request'); // npm install request
88 var printdata = function(apiurl) {
89     request(apiurl, function (error, response, body) {
90         if (!error && response.statusCode == 200) {
91             data = JSON.parse(body);
92             docs2console(data2docs(data));
93         }
94     });
95 };
96
97 /* This produces the following output:
98 Date: Fri May 23 2003 10:00:00 GMT-0700 (PDT)
99 Title: National Security Agency/Central Security Service (NSA/CSS) Freedom of Information Act
100
101 Date: Fri Dec 14 2012 09:00:00 GMT-0800 (PST)
102 Title: Human Rights Day and Human Rights Week, 2012
103
104 Date: Mon Oct 20 2008 10:00:00 GMT-0700 (PDT)
105 Title: Tarp Capital Purchase Program
106 */

```

Note several things that we did here:

- In the last example, we added in an explicit time and timezone when converting the string timestamps into Date objects. This is for reproducibility; if you don't specify this then JS will use the local timezone.

- The number of milliseconds since the Unix Epoch is a bit hard to verify at first glance. We defined some subroutines which use the fact that there are 86400 seconds per day to do some quick sanity checks.
- Differences between Dates are not Dates themselves, and you have to be careful about working with them. There are [modules](#) that can help with this, of which [moment.js](#) is the most popular.

You don't need to memorize all the Date methods; just know that you probably want to convert timestamps into Date instances before doing things with them, and also make heavy use of the REPL when working with Dates to confirm that your code is not automatically guessing times or timezones against your intention.

RegExp

The [RegExp](#) object is used to specify regular expressions. It uses Perl-inspired regular expressions (though [not](#) full PCRE) and can be used to recognize patterns in strings as well as to do basic parsing and extraction. Here are some examples⁵ of its use:

```

1  // Regexp examples
2
3  // 1. Test for presence, identify location
4  var ex1 = "The quick brown fox jumped over the lazy dogs.";
5  var re1 = /(cr|l)azy/;
6  console.log(re1.test(ex1));
7  // true
8  console.log(re1.exec(ex1));
9  /*
10 [ 'lazy',
11   'l',
12   index: 36,
13   input: 'The quick brown fox jumped over the lazy dogs.' ]
14 */
15
16
17 // 2. Global matches
18 var ex2 = "Alpha Beta Gamma Epsilon Omicron Theta Phi";
19 var re2 = /(\w+a )/;
20 var re2g = /(\w+a )/g;
21
22 re2.exec(ex2);
23 re2.exec(ex2);
24 /* Without the /g, repeating the match doesn't do anything here.
25
26 [ 'Alpha ',
27   'Alpha ',

```

⁵Note that in the very last example, we make use of some of the methods from the [underscore.js](#) library. Read the section on JS Functions for more on this.

```

28     index: 0,
29     input: 'Alpha Beta Gamma Epsilon Omicron Theta Phi' ]
30
31 [ 'Alpha ',
32   'Alpha ',
33   index: 0,
34   input: 'Alpha Beta Gamma Epsilon Omicron Theta Phi' ]
35 */
36
37 re2g.exec(ex2);
38 re2g.exec(ex2);
39 re2g.exec(ex2);
40 re2g.exec(ex2);
41 re2g.exec(ex2);
42 /* With the /g, repeating the match iterates through all matches.
43
44 [ 'Alpha ',
45   'Alpha ',
46   index: 0,
47   input: 'Alpha Beta Gamma Epsilon Omicron Theta Phi' ]
48
49 [ 'Beta ',
50   'Beta ',
51   index: 6,
52   input: 'Alpha Beta Gamma Epsilon Omicron Theta Phi' ]
53
54 [ 'Gamma ',
55   'Gamma ',
56   index: 11,
57   input: 'Alpha Beta Gamma Epsilon Omicron Theta Phi' ]
58
59 [ 'Theta ',
60   'Theta ',
61   index: 33,
62   input: 'Alpha Beta Gamma Epsilon Omicron Theta Phi' ]
63
64 null
65 */
66
67 // We can formalize the process of iterating through the
68 // matches till we hit a null with the following function:
69 var allmatches = function(rex, str) {
70     var matches = [];
71     var match;
72     while(true) {
73         match = rex.exec(str);
74         if(match !== null) { matches.push(match); }

```

```

75         else { break; }
76     }
77     return matches;
78 };
79 allmatches(re2g, ex2);
80 /*
81 [ [ 'Alpha ',
82     'Alpha ',
83     index: 0,
84     input: 'Alpha Beta Gamma Epsilon Omicron Theta Phi' ],
85   [ 'Beta ',
86     'Beta ',
87     index: 6,
88     input: 'Alpha Beta Gamma Epsilon Omicron Theta Phi' ],
89   [ 'Gamma ',
90     'Gamma ',
91     index: 11,
92     input: 'Alpha Beta Gamma Epsilon Omicron Theta Phi' ],
93   [ 'Theta ',
94     'Theta ',
95     index: 33,
96     input: 'Alpha Beta Gamma Epsilon Omicron Theta Phi' ] ]
97 */
98
99
100 // 3. Case-insensitive
101 var ex3 = "John is here.";
102 var re3s = /JOHN/;
103 var re3i = /JOHN/i;
104 re3s.test(ex3); // false
105 re3i.test(ex3); // true
106
107
108 // 4. Multiline
109 var ex4 = "Alpha beta gamma.\nAll the king's men.\nGermany France Italy.";
110 var re4s = /^G/;
111 var re4m = /^G/m;
112 re4s.test(ex4); // false
113 re4m.test(ex4); // true
114
115
116 // 5. Parsing postgres URLs
117 var pgurl = "postgres://myuser:mypass@example.com:5432/mydbpass";
118 var pgregex = /postgres:\/\/([~:]+):([~@]+)@([~:]+):(\d+)\.\/(.+)/;
119 var flag = pgregex.test(pgurl); // true
120 var out = pgregex.exec(pgurl);
121 /*

```



```

7
8  /*
9  [ 'E',
10    'LN10',
11    'LN2',
12    'LOG2E',
13    'LOG10E',
14    'PI',
15    'SQRT1_2',
16    'SQRT2',
17    'random',
18    'abs',
19    'acos',
20    'asin',
21    'atan',
22    'ceil',
23    'cos',
24    'exp',
25    'floor',
26    'log',
27    'round',
28    'sin',
29    'sqrt',
30    'tan',
31    'atan2',
32    'pow',
33    'max',
34    'min' ]
35  */
36
37
38  // 2. Generating random integers between a and b, not including the upper bound.
39  var randint = function(a, b) {
40      var frac = Math.random();
41      return Math.floor((b-a)*frac + a);
42  };
43
44
45  // 3. Recapitulating constants
46  Math.pow(Math.E, Math.LN2); // 1.9999999999999998
47  Math.pow(Math.E, Math.LN10); // 10.000000000000002
48  Math.pow(10, Math.LOG10E) - Math.E; // 0
49  Math.pow(2, Math.LOG2E) - Math.E; // 0
50
51
52  // 4. Determining the effective growth rate from the start time (in months),
53  // the stop time (in months), the initial user base, and the final user

```

```

54 // base.
55 //
56 //  $init * 2^{\{(stop-start)/tau\}} = fin$ 
57 //
58 //  $tau = (stop-start)/log_2(fin/init)$ 
59 //
60 var log2 = function(xx) {
61     return Math.log(xx)/Math.LN2;
62 };
63
64 var doublingtime = function(start, stop, init, fin) {
65     var dt = (stop-start);
66     var fold = fin/init;
67     return dt/log2(fold);
68 };
69
70 log(doublingtime(0, 10, 1, 16)); // 2.5
71 var tau = doublingtime(0, 24, 1, 9); // 7.571157042857489
72 Math.pow(2, 24/tau); // 9.000000000000002

```

For the basics this is fine, but in practice you probably don't want to do too much math⁶ in JS. If you have heavily mathematical portions of your code, you might be able to implement them in Python via the [numpy](#) and [scipy](#) libraries (or the new [blaze](#)) and expose them over a [simple webservice](#) if you can tolerate the latency of an HTTP request. The other alternative is to implement your numerical code in C or C++ and then link it into JS via the built-in capability for C/C++ [addons](#) or a convenience library like the node foreign function interface ([node-ffi](#)).

String

The [String](#) object provides a few basic methods for string manipulation:

```

1 // String examples
2
3 // 1. Treat String as Array of characters
4 var log = console.log;
5 var sx1 = "The quick brown fox jumped over the lazy dogs.";
6 sx1.charAt(10); // 'b'
7 sx1.slice(10, 13); // 'bro'
8 sx1.substring(10, 13); // 'bro'
9 sx1.substr(10, 13); // 'brown fox jum'

```

⁶If you are interested in rendering math with JS, look at the powerful [MathJax](#) library for generating L^AT_EX in the browser. For server-side math, the new [mathjs](#) library might be worth checking out, as are the [matrix](#) libraries. However, math in node is very much in its infancy and numerical linear algebra is hard, subtle, and highly [architecture dependent](#). So you probably want to rely on a heavily debugged external library like Python's [numpy](#) or the [GNU Scientific Library \(GSL\)](#) codebase, and then bridge it into node via a system-call, a webservice, the built-in provision for addons, or a foreign-function interface library (see text).

```

10 | sx1.length; // 46
11 |
12 | // 2. Compare strings using alphabetical ordering
13 | var sx2 = "alpha";
14 | var sx3 = "beta";
15 | log(sx2 < sx3); // true
16 |
17 | // 3. Replace substrings via string or regex
18 | var sx4 = sx2.replace('ph', 'foo');
19 | log(sx2); // 'alpha'
20 | log(sx4); // 'alfooa'
21 |
22 | var sx5 = sx2.replace(/a$/, 'bar'); // NOTE regex
23 | log(sx2); // 'alpha'
24 | log(sx4); // 'alphbar'
25 |
26 | // 4. Change case (lower, upper)
27 | log(sx2.toUpperCase()); // 'ALPHA'
28 |
29 | // 5. Trim strings
30 | var sx6 = " " + ['Field1', 'Field2', 'Field3'].join("\t") + "\n";
31 | var sx7 = sx6.trimRight();
32 | var sx8 = sx6.trimLeft();
33 | var sx9 = sx6.trim();
34 | log(sx6); // ' Field1\tField2\tField3\n'
35 | log(sx7); // ' Field1\tField2\tField3'
36 | log(sx8); // 'Field1\tField2\tField3\n'
37 | log(sx9); // 'Field1\tField2\tField3'
38 |
39 | // 6. Split strings (useful for simple parsing)
40 | var fields = sx9.split("\t");
41 | log(fields); // [ 'Field1', 'Field2', 'Field3' ]

```

Again, for the basics this is fine, but for extremely heavy string manipulation you can lean on a library like the ones listed [here](#), particularly the popular [underscore-string](#).

JSON

The [JSON](#) global is used for rapidly parsing Javascript Object Notation (JSON), a subset⁷ of JS used for serializing data to disk and communicating between programming languages. JSON has [replaced](#) XML for most new applications because it's more human-readable than XML, easier to parse, doesn't require an (often-missing) separate [DTD](#) file to interpret the document, and has wide language support.

⁷If you want to be a language lawyer, JSON is not [technically](#) a strict subset of JS. However, for most purposes it can be treated as such.

```

1 // JSON examples
2
3 // 1. List methods on JSON object
4 var log = console.log;
5 Object.getOwnPropertyNames(JSON) // [ 'parse', 'stringify' ]
6
7 // 2. Using JSON.parse to deserialize JSON strings.
8 //
9 //     Note that you use double quotes within JSON and single quotes to
10 //     encapsulate the entire string
11 var jsdata = '[{"asdf":9, "bar":10}, 18, "baz"]';
12 var data = JSON.parse(jsdata);
13 log(data[0].asdf); // 9
14
15 // You can also do this with eval. But don't do that. Use JSON.parse instead.
16 var data2 = eval(jsdata);
17 log(data2[0].asdf); // 9
18
19 // 3. Using JSON.stringify to serialize JS objects.
20 //
21 //     While strings, numbers, arrays, and dictionaries/objects are generally
22 //     safe, note that Regexp instances don't have a good default
23 //     serialization and thus need special handling.
24 var dt = new Date('2003-05-23T17:00:00Z');
25 var rex = /(cr|l)/;
26 var data3 = [9, {"foo": dt, "bar": rex, "baz": {"quux": "a", "alpha": [77, 3] }}, 11];
27 log(data3);
28 /*
29 [ 9,
30   { foo: Fri May 23 2003 10:00:00 GMT-0700 (PDT),
31     bar: /(cr|l)/,
32     baz: { quux: 'a', alpha: [Object] } },
33   11 ]
34 */
35
36 // Note that the Regexp instance is serialized to {} rather than "/(cr|l)/"
37 var data3str = JSON.stringify(data3);
38 // '[9,{"foo":"2003-05-23T17:00:00.000Z","bar":{},"baz":{"quux":"a","alpha":[77,3]}},11]'
39
40 // We can restore the data structure as shown. Note again the the restoration
41 // is only as good as the serialization. Make sure to look at the raw JSON string
42 // output
43 var data4 = JSON.parse(data3str);
44 log(data4);
45 /*
46 [ 9,

```



```

47 { foo: '2003-05-23T17:00:00.000Z',
48   bar: {},
49   baz: { quux: 'a', alpha: [Object] } },
50 11 ]
51 */

```

As a rule of thumb, you can consider using XML instead of JSON if you are actually marking up a document, like a novel or a newspaper article. But for other data interchange purposes you should usually use JSON.

Error

The [Error](#) object is used for exception handling; see in particular MDN's page on [throw](#) for some good examples related to invalid input types, along with the idea of rethrowing an exception.

```

1  // Error examples
2
3  // 1. List all methods in Error
4  Object.getOwnPropertyNames(Error);
5  /*
6  [ 'length',
7    'name',
8    'arguments',
9    'caller',
10   'prototype',
11   'captureStackTrace',
12   'stackTraceLimit' ]
13  */
14
15 // 2. An example of try/catch
16 var log = console.log;
17 var div = function(a, b) {
18   try {
19     if(b === 0) {
20       throw new Error("Divided by Zero");
21     } else {
22       return a/b;
23     }
24   } catch(e) {
25     log('name\n%s\n\nmessage\n%s\n\nstack\n%s', e.name, e.message, e.stack);
26   }
27 };
28
29 /*
30 name
31 Error

```

```

32
33 message
34 undefined
35
36 stack
37 Error: Divided by Zero
38   at div (repl:1:79)
39   at repl:1:2
40   at REPLServer.self.eval (repl.js:112:21)
41   at Interface.<anonymous> (repl.js:239:12)
42   at Interface.EventEmitter.emit (events.js:95:17)
43   at Interface._onLine (readline.js:202:10)
44   at Interface._line (readline.js:531:8)
45   at Interface._ttyWrite (readline.js:767:16)
46   at ReadStream.onkeypress (readline.js:99:10)
47   at ReadStream.EventEmitter.emit (events.js:98:17)
48 */
49
50
51 // 3. Returning an Error object directly
52 var div2 = function(a, b) {
53   if(b === 0) {
54     return new Error("Divided by Zero");
55   } else {
56     return a/b;
57   }
58 };
59 var err = div2(4, 0);
60 log(err.stack);
61 /*
62 Error: Divided by Zero
63   at div2 (repl:1:62)
64   at repl:1:11
65   at REPLServer.self.eval (repl.js:112:21)
66   at repl.js:249:20
67   at REPLServer.self.eval (repl.js:122:7)
68   at Interface.<anonymous> (repl.js:239:12)
69   at Interface.EventEmitter.emit (events.js:95:17)
70   at Interface._onLine (readline.js:202:10)
71   at Interface._line (readline.js:531:8)
72   at Interface._ttyWrite (readline.js:767:16)
73 */
74
75
76 // 4. Using custom error types.
77 //
78 // Modified from Zip Code example here:

```

```

79 // https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/throw#Example
80 //
81 // Note that for a real email parser, you will want to use an existing
82 // regex rather than writing your own, as email formats can get
83 // surprisingly complex. See RFC 2822 and here:
84 //
85 // http://www.regular-expressions.info/email.html
86 // http://tools.ietf.org/html/rfc2822#section-3.4.1
87 //
88 // Better yet, if you really want to diagnose problems with invalid email
89 // addresses use a full function that returns specific errors for different
90 // cases. See here: stackoverflow.com/q/997078
91
92 var Email = function(emstr) {
93     var regex = /^[^@+](.*)@([^\.\.]+\.)?([^\.\.]+\.)?$/;
94     if (regex.test(emstr)) {
95         var match = regex.exec(emstr);
96         this.user = match[1];
97         this.domain = match[2];
98         this.tld = match[3];
99         this.valueOf = function() {
100             return this.value;
101         };
102         this.toString = function() {
103             return this.user + '@' + this.domain + '.' + this.tld;
104         };
105     } else {
106         throw new EmailFormatException(emstr);
107     }
108 };
109
110 var EmailFormatException = function(value) {
111     this.value = value;
112     this.message = "not in a@b.c form";
113     this.toString = function() {
114         return this.value + this.message;
115     };
116 };
117
118 var EMAIL_INVALID = -1;
119 var EMAIL_UNKNOWN = -1;
120
121 var parseEmail = function(instr) {
122     try {
123         em = new Email(instr);
124     } catch (e) {
125         if (e instanceof EmailFormatException) {

```

```

126         return EMAIL_INVALID;
127     } else {
128         return EMAIL_UNKNOWN_ERROR;
129     }
130 }
131 return em;
132 };
133
134 // Make sure to include new when using the Email constructor directly
135 var foo = Email('john@gmail.com'); // Doesn't work
136 var bar = new Email('joe@gmail.com'); // Works
137
138 // Here's what happens when we invoke parseEmail
139 parseEmail('john@gmailcom'); // -1 == EMAIL_INVALID
140 parseEmail('john@gmail.com'); // -1 == EMAIL_INVALID
141 parseEmail('john@gmail.com');
142 /*
143 { user: 'john',
144   domain: 'gmail',
145   tld: 'com',
146   valueOf: [Function],
147   toString: [Function] }
148 */

```

The try/catch paradigm is frequently used in browser JS, and you should be aware of it, but for node's callback heavy environment it's not ideal. We'll cover more on errors and exception in node specifically later. Be aware though that the `Error.stack` field is only available in IE10 and later (see [here](#) and [here](#)), and that you'll generally want to return instances of Error objects rather than using try/catch in node.

Built-in functions

In addition to these globals (Array, Date, RegExp, Math, String, JSON, Error), here are several miscellaneous built-in functions that come in for frequent use.

```

1 // Built-in examples
2
3 /*
4   1. decodeURI and encodeURI: escape special characters in URIs
5
6   These two functions ensure that special characters like brackets and
7   slashes are escaped in URI strings. Let's illustrate with an
8   example API call to the federalregister.gov site.
9 */
10 var apiurl = "https://www.federalregister.gov/api/v1/articles/" +
11 "03-12969,2012-30312,E8-24781.json?fields%5B%5D=title" +
12 "&fields%5B%5D=publication_date";

```

```

13 decodeURI(apiurl);
14 // 'https://www.federalregister.gov/api/v1/articles/03-12969,
15 // 2012-30312,E8-24781.json?fields[]=title&fields[]=publication_date'
16 encodeURI(decodeURI(apiurl));
17 // 'https://www.federalregister.gov/api/v1/articles/03-12969,
18 // 2012-30312,E8-24781.json?fields%5B%5D=title&fields%5B%5D=publication_date'
19
20 /*
21 2. typeof: gives the type of a JS variable.
22
23 Note: typeof cannot be used by itself, as it won't distinguish between
24 many kinds of objects (see here: http://stackoverflow.com/a/7086473). It
25 is, however, used by many libraries in conjunction with instanceof and
26 various kinds of duck-typing checks. See here:
27 tobyho.com/2011/01/28/checking-types-in-javascript
28
29 In general, metaprogramming in JS without a reliable library (something
30 like modernizr.js, but for types rather than browser features) is a pain
31 if you want to do it in full generality; it's not as regular as Python.
32 */
33
34 typeof 19
35 // 'number'
36 typeof "foo"
37 // 'string'
38 typeof {}
39 // 'object'
40
41 /*
42 3. parseInt and parseFloat: convert strings to ints and floats
43
44 Note that parseInt takes a base as the second argument, and can be used
45 to convert up to base 36.
46 */
47
48 parseInt('80', 10) // = 8*101 + 0*100
49 // 80
50 parseInt('80', 16) // = 8*161 + 0*160
51 // 128
52 parseFloat('89803.983')
53 // 89803.983
54
55 /*
56 4. Object.getOwnPropertyNames
57
58 Useful routine that can remind you which methods exist on a particular
59 object.

```

```

60  */
61
62  Object.getOwnPropertyNames(JSON)
63  // [ 'parse', 'stringify' ]
64
65
66  /*
67   5. Object.getOwnPropertyDescriptor
68
69   Use this to introspect the fields of a given object or module.
70  */
71
72  var uu = require('underscore');
73  Object.getOwnPropertyDescriptor(uu, 'map');
74  /*
75  { value: [Function],
76    writable: true,
77    enumerable: true,
78    configurable: true }
79  */

```

You can see a full list of [globals and builtins](#) at MDN; note in particular which ones are non-standard and which are available in all modern JS interpreters.

Functional Programming (FP) and JS Functions

Now that you have some familiarity with the built-ins, let's go through the various tricks and conventions related to JS functions, as well as several examples of doing functional programming in JS.

First-class functions

The concept of first-class functions (i.e. functions themselves as arguments and variables) is key to understanding modern JS. We've been using these for a while in different ways, but let's go through an example where we compare and contrast the functional programming (FP) style to the procedural style:

```

1  #!/usr/bin/env node
2
3  // Implementation 1: Procedural
4  var log = console.log;
5  var n = 5;
6  var out = [];
7  for(var i = 0; i < n; i++) {
8      out.push(i * i);
9  }
10 log(out); // [ 0, 1, 4, 9, 16 ]

```

```

11
12 // Implementation 2: Functional Programming
13 // Functions as first class variables
14 //
15 // Allows us to abstract out the loop and the function applied within, such
16 // that we can swap in a new function or a new kind of iteration.
17 var sq = function(x) { return x * x;};
18 var cub = function(x) { return x * x * x;};
19 var loop = function(n, fn) {
20     var out = [];
21     for(var i = 0; i < n; i++) {
22         out.push(fn(i));
23     }
24     return out;
25 };
26 var loopeven = function(n, fn) {
27     var out = [];
28     for(var i = 0; i < n; i++) {
29         if(i % 2 === 0) {
30             out.push(fn(i));
31         } else {
32             out.push(i);
33         }
34     }
35     return out;
36 };
37 log(loop(n, sq));      // [ 0, 1, 4, 9, 16 ]
38 log(loop(n, cub));    // [ 0, 1, 8, 27, 64 ]
39 log(loopeven(n, sq)); // [ 0, 1, 4, 3, 16 ]
40 log(loopeven(n, cub)); // [ 0, 1, 8, 3, 64 ]
41
42
43 // Implementation 3: Functional Programming with underscorejs.org
44 // Note the use of the map and range methods.
45 // - range: generate an array of numbers between 0 and n
46 // - map: apply a function to a specified array
47 var uu = require('underscore');
48 log(uu.map(uu.range(0, n), sq)); // [ 0, 1, 4, 9, 16 ]
49 log(uu.map(uu.range(0, n), cub)); // [ 0, 1, 8, 27, 64 ]

```

By using so-called “higher order” functions (i.e. functions that accept other functions as arguments), we have cleanly decoupled the internal function and the external loop in this simple program. This is actually a very common use case. You may want to replace the loop with a graph traversal, an iteration over the rows of a matrix, or a recursive descent down the leaves of a binary tree. And you may want to apply an arbitrary function during that traversal. This is but one of many situations in which FP provides an elegant solution.

Functional programming and underscore.js

You can certainly do functional programming with the standard JS primitives, but it is greatly facilitated by the important [underscore.js](#) library. Let's redo the first Project Euler problem by making heavy use of underscore:

```
1 #!/usr/bin/env node
2 // http://projecteuler.net/problem=1
3 // If we list all the natural numbers below 10 that are multiples of 3 or 5,
4 // we get 3, 5, 6 and 9. The sum of these multiples is 23.
5 //
6 // Find the sum of all the multiples of 3 or 5 below 1000.
7 //
8 // We'll make heavy use of underscore here to illustrate some FP techniques.
9 var uu = require('underscore');
10
11 // For the anydivide function
12 // - uu.map takes an array and a univariate function, and applies the
13 //   function element-wise to the array, returning a new array.
14 //
15 // - We define divfn via a closure, so that it takes one variable and
16 //   returns one output. This can then be passed to uu.map.
17 //
18 // - uu.any takes an array of booleans and returns true if and only if one
19 //   of them is true, and false otherwise.
20 var anydivide = function(as, b) {
21     var divfn = function(a) { return b % a === 0; };
22     return uu.any(uu.map(as, divfn));
23 };
24
25 // For the sum function
26 //
27 // - uu.reduce takes an array and a function, and applies that function
28 //   iteratively to each element of the array, starting from the first element.
29 // - The anonymous function we pass in as the second argument just adds adjacent
30 //   elements together.
31 //
32 var sum = function(arr) {
33     return uu.reduce(arr, function(a, b) { return a + b; });
34 };
35
36 // For the fizzbuzz function
37 //
38 // - We define divfn as a closure, using the factors variable. Again, this
39 //   is so that it takes one argument, returns one output, and can be passed to map.
40 //
41 // - We use uu.filter to iterate over an array and return all elements
42 //   which pass a condition, specified by a function that returns booleans.
```



```

43 // The array is the first argument to uu.filter and the boolean-valued
44 // func is the second argument.
45 //
46 // - We use uu.range to generate an array of numbers between two specified
47 // - values.
48 //
49 var fizzbuzz = function(factors, max) {
50     var divfn = function(nn) { return anydivide(factors, nn); };
51     var divisible = uu.filter(uu.range(1, max), divfn);
52     return sum(divisible);
53 };
54
55 // console.log(fizzbuzz([3, 5], 10));
56 console.log(fizzbuzz([3, 5], 1000));

```

We've commented this much more heavily than we normally would. You can explore the full panoply of underscore methods [here](#). While the functional style might be a bit hard to understand at first, once you get the hang of it you'll find yourself using it whenever iterating over arrays, applying functions to arrays, generating new arrays, and the like. It makes your code more compact and the systematic use of first-class functions greatly increases modularity. In particular, by using the FP paradigm you can often⁸ reduce the entire internals of a program to the application of a single function to an external data source, such as a text file or a JSON stream.

Partial functions and the concept of currying

A common paradigm in functional programming is to start with a very general function, and then set several of the input arguments to default values to simplify it for first-time users or common situations. We can accomplish this via currying, as shown:

```

1 // 1. Partial function application in JS
2 //
3 // The concept of partial function application (aka currying, named after
4 // Haskell Curry the mathematician).
5 var sq = function(x) { return x * x; };
6 var loop = function(n, fn) {
7     var out = [];
8     for(var i = 0; i < n; i++) {
9         out.push(fn(i));
10    }
11    return out;
12 };
13

```

⁸Note that the asynchronous nature of node might seem to interfere with the idea of expressing the whole thing as $f(g(h(x)))$, as it seems instead that you'd have to express it as $h(x, g(y, f))$ via callbacks. However, as we will see, with the appropriate flow control libraries and especially the pending introduction of the `yield` keyword in node, it again becomes feasible to code in a functional style.

```

14 // We can use the 'bind' method on Functions to do partial evaluation
15 // of functions, as shown.
16 //
17 // Here, loopN now accepts only one argument rather than two. (We'll cover
18 // the null argument passed in to bind when we discuss 'this').
19 loop(10, sq); // [ 0, 1, 4, 9, 16, 25, 36, 49, 64, 81 ]
20 var loopN = loop.bind(null, 10); // n = 10
21 loopN(sq); // [ 0, 1, 4, 9, 16, 25, 36, 49, 64, 81 ]
22
23 // 2. Implementing currying in underscore is easy with the 'partial' method.
24 var uu = require('underscore');
25 loop6 = uu.partial(loop, 6);
26 loop6(sq); // [ 0, 1, 4, 9, 16, 25 ]

```

Note that underscore provides tools for doing this as well. In general, if you want to do something in JS in the FP style, underscore is a very powerful tool.

Function scope

The scope where variables are defined is a little tricky in JS. By default everything is global, which means that you can easily have a namespace collision between two different libraries. There are many gotchas in this area, but you can generally get away with considering the following four ways of thinking about scope.

- global scope
- statically defined functions with statically defined scope
- dynamically defined functions with statically defined scope (closures)
- dynamically defined functions with dynamically defined scope (**this**)

Here, we use the term “statically defined” to indicate that the function or scope can be fully specified simply by inspecting the `.js` file (“static analysis”), whereas “dynamically defined” means that the function or scope can take on different values in response to user input, external files, random number generation, or other external variables that are not present in the source code `.js` file. Let’s see some examples of each of these:

```

1 // Scope in JS
2
3 // 1. Unintentional global variables
4 //
5 // This can occur if you forget to declare cc within foo.
6 var cc = 99;
7 var foo = function(aa) {
8     var bb = 7;
9     console.log("aa = " + aa);
10    console.log("bb = " + bb);

```

```

11     console.log("cc = " + cc);
12 };
13 foo(22);
14 /*
15 aa = 22
16 bb = 7
17 cc = 99
18 */
19
20 // 2. Intentional global variables
21 //
22 //     This is the recommended syntax within node if you really
23 //     want to use globals. You can accomplish the same thing with
24 //     window in the browser.
25 var dd = 33;
26 var bar = function(aa) {
27     var bb = 7;
28     console.log("aa = " + aa);
29     console.log("bb = " + bb);
30     console.log("dd = " + global.dd);
31 };
32 bar(22);
33 /*
34 aa = 22
35 bb = 7
36 dd = 33
37 */
38
39
40 // 3. Scope in statically defined functions
41 //
42 //     The aa defined within this functions' argument and the bb within its
43 //     body override the external aa and bb. This function is defined
44 //     'statically', i.e. it is not generated on the fly from input arguments
45 //     while the program is running.
46 var aa = 7;
47 var bb = 8;
48 var baz = function(aa) {
49     var bb = 9;
50     console.log("aa = " + aa);
51     console.log("bb = " + bb);
52 };
53 baz(16);
54 /*
55 aa = 16
56 bb = 9
57 */

```

```

58
59
60 // 4. Scope in dynamically defined functions (simple)
61 //
62 //     Often we'll want to build a function on the fly, in which case the use
63 //     of variables from the enclosing scope is a feature rather than a
64 //     bug. This kind of function is called a closure. Let's first do a
65 //     trivial example.
66 //
67 //     Here, buildfn is intentionally using the increment variable
68 //     passed in as an argument in the body of the newfn, which is
69 //     being created on the fly.
70 var buildfn = function(increment) {
71     var newfn = function adder(xx) { return xx + increment; };
72     return newfn;
73 };
74 buildfn(10); // [Function: adder]
75 buildfn(10)(17); // 27
76 var add3 = buildfn(3);
77 var add5 = buildfn(5);
78 add3(add5(17)); // 25
79
80
81 //     Here's a more complex example where we pass in a function rather
82 //     than simply assuming buildfn2 will be doing addition.
83 var buildfn2 = function(yy, binaryoperator) {
84     var cc = 10;
85     var newfn = function binop(xx) { return cc*binaryoperator(xx, yy); };
86     return newfn;
87 };
88 var pow = function(aa, bb) { return Math.pow(aa, bb);}
89 var fn = buildfn2(3, pow);
90 fn(5); // 10 * Math.pow(5, 3)
91
92
93 // 5. Scope in dynamically defined functions (more advanced)
94 //
95 //     Here's a more realistic example, where we define a meta-function that
96 //     takes two other functions and snaps them together to achieve an
97 //     effect.
98 var compose = function(f, g) {
99     var h = function(x) {
100         return f(g(x));
101     };
102     return h;
103 };
104

```

```

105 var jsdata = '[{"asdf":9, "bar":10}, 18, "baz"]';
106 var f1 = function(data) { return data[0].bar + 11;};
107 var f2 = JSON.parse;
108 f1(f2(jsdata)); // 10 + 11 === 21
109 var f1f2 = compose(f1, f2);
110 f1f2(jsdata); // 10 + 11 === 21
111
112
113 // 6. Dynamically defined scope in functions (this)
114 //
115 //     Now, what if we want to go one level more abstract? This time
116 //     we want to define a closure that doesn't take on its values
117 //     from the current context, but from some context that we will
118 //     dynamically specify at a later date.
119 var bb = 10;
120 var static_closure = function(aa) {
121     return aa + bb;
122 };
123 static_closure(3);
124 // 13
125
126 var dynamic_closure = function(aa) {
127     return aa + this.bb;
128 };
129
130 var context1 = {
131     'fn': dynamic_closure,
132     'bb': 10
133 };
134
135 context1.fn(3);
136 // 13
137
138 var context2 = {
139     'fn': dynamic_closure,
140     'bb': Math.random()
141 };
142
143 context2.fn(3);
144 // 3.656272369204089 [your value will be different]
145
146
147 //
148 // Great. So, to recap, the 'this' variable points to a single object or
149 // else is undefined. You can use it to leave the enclosing scope for a
150 // closure unspecified until the last moment when it's executed.
151 //

```

```

152 // If you can, you may want to simply pass in the object itself as an
153 // explicit variable, as shown below. Otherwise one can use 'this'.
154 var simpler_than_dynamic_closure = function(aa, obj) {
155     return aa + obj.bb;
156 };
157
158 simpler_than_dynamic_closure(3, context2);
159 // 3.656272369204089

```

This example shows:

- how one can inadvertently use globals
- how to use globals intentionally if you really need to
- how scope works within standard statically defined functions
- how you can dynamically define functions via closures.
- how you can even dynamically define enclosing scope via the `this` keyword

While there are definitely times you do want to build closures, you don't want to inadvertently use global variables locally (or, more generally, have a scoping bug). There are three ways to ensure that this doesn't happen. First, invoke JSHint as described in Lecture 4b to confirm that all your variables have `var` in front of them; you can set up a `~/jshintrc` to assist with this (see [here](#)). Second, use the node and Chrome JS REPLs aggressively to debug sections of code where you're unsure about the scope; you may need to do this quite a bit to understand the `this` keyword at the beginning. Third, you can and should aggressively encapsulate your code.

Encapsulation: the self-executing function trick

A very common idiom you will see in JS code is the self-executing function trick. We define a function and then immediately execute it, returning a single object that contains the results of that function:

```

1 // The concept of self-executing functions
2
3 // 1. Basic self-executing functions for encapsulation
4 //
5 // Suppose you have some JS code like this:
6 var foo = "inner";
7 console.log(foo);
8
9 // You worry that the external world may also have defined a foo variable.
10 // To protect your code, you can encapsulate it by putting it in a
11 // self-executing function.
12 var foo = "outer";
13 (function() {

```

```

14     var foo = "inner";
15     console.log(foo);
16 }());
17 console.log(foo);
18
19 // This will print:
20 /*
21 inner
22 outer
23 */
24
25 // Let's break this down bit by bit. The inside is just an anonymous
26 // function definition. Let's assign it to a variable.
27 var anon = function () {
28     var foo = "inner";
29     console.log(foo);
30 };
31
32 // We enclose this function definition in parentheses, and then immediately
33 // execute it by adding another two parens at the end. We also need a
34 // semicolon as the expression is ending.
35 (function() {
36     var foo = "inner";
37     console.log(foo);
38 })();
39
40 // 2. Passing in variables to a self-executing function
41 //
42 // Now that we've set up a wall, we can do something more sophisticated by
43 // passing in specific variables from the enclosing scope.
44 var bar = "allowed";
45 (function(aa) {
46     var foo = "inner";
47     console.log(foo + " " + aa);
48 })(bar);
49
50 // This prints
51 /*
52 inner allowed
53 */
54
55 // 3. Passing in and receiving variables from a self-executing function.
56 //
57 // Now we'll return an object as well, converting the inside into a proper
58 // function.
59 var bar = "allowed";
60 var result = (function(aa) {

```

```

61     var foo = "inner";
62     var out = foo + " " + aa;
63     console.log(out);
64     return {"val": out};
65 })(bar);
66 console.log(result);
67
68 // This prints
69 /*
70 { val: 'inner allowed' }
71 */

```

This is a hack to compensate for the fact that JS, unlike (say) Python or Ruby, did not have a builtin module system. Today the combination of the node `require` syntax and npm gives a good solution for server-side modules, while browserify is a reasonable way of porting these into client-side modules (though development of modules within the ECMAScript standard continues apace).

Callbacks, asynchronous functions, and flow control

Now that you have a deeper understanding of the concept of first-class functions and functional programming, we can explain the concept of callbacks and then asynchronous functions. A callback is a function that is passed in as an argument to another function. When the second function completes, it feeds the results that would normally be returned directly into the arguments of the callback. Let's look at an example:

```

1  #!/usr/bin/env node
2
3  var request = require('request'); // npm install request
4  var apiUrl = 'http://nodejs.org/api/index.json';
5  var callbackfn = function(error, response, body) {
6      if (!error && response.statusCode == 200) {
7          console.log(body);
8      }
9  };
10 request(apiUrl, callbackfn);
11
12 // When executed, this produces the following
13 /*
14 {
15     "source": "doc/api/index.markdown",
16     "desc": [
17         {
18             "type": "list_start",
19             "ordered": false
20         },
21         {

```



```

22     "type": "list_item_start"
23 },
24 {
25     "type": "text",
26     "text": "[About these Docs](documentation.html)"
27 },
28 {
29     "type": "list_item_end"
30 },
31 {
32     "type": "list_item_start"
33 },
34 {
35     "type": "text",
36     "text": "[Synopsis](synopsis.html)"
37 },
38 {
39     "type": "list_item_end"
40 },
41 {
42     "type": "list_item_start"
43 },
44 ...
45 */

```

We may want to pass callbacks into callbacks. For example, suppose that we want to write the results of this API call to disk. Below is the messy way to do this. This works, but it is bad in a few respects. First, it's difficult to read. Second, it mixes three different functions together (making the HTTP request, extracting the body from the HTTP response, and writing the body to disk). Third, it is hard to debug and test in isolation. Fourth, because the default output of `request` that is passed to the callback is the 3-tuple of (`error`, `response`, `body`), we're doing a bit of a hack to pass `outfile` through to the second and third parts. The messy entanglement below is what is known as “callback hell”.

```

1  #!/usr/bin/env node
2
3  var request = require('request');
4  var apiurl = 'http://nodejs.org/api/index.json';
5  var fs = require('fs');
6  var outfile = 'index2.json';
7  request(apiurl, function(error, response, body) {
8      if (!error && response.statusCode == 200) {
9          fs.writeFile(outfile, body, function (err) {
10             if (err) throw err;
11             var timestamp = new Date();
12             console.log("Wrote %s %s", outfile, timestamp);
13         });

```

```
14     }
15 });
```

Now, there is one advantage of the callback approach, which is that by default it supports asynchronous execution. In this version we have wrapped the request in `setTimeout` just to show that even though the request is executed first, it runs in the background asynchronously. Code that is executed after the request can complete before it. This is the essence of asynchronous execution: a line of code does not wait for preceding lines of code to finish unless you explicitly force it to be synchronous.

```
1 #!/usr/bin/env node
2
3 var request = require('request');
4 var apiurl = 'http://nodejs.org/api/index.json';
5 var fs = require('fs');
6 var outfile = 'index3.json';
7 setTimeout(function() {
8     request(apiurl, function(error, response, body) {
9         if (!error && response.statusCode == 200) {
10             fs.writeFile(outfile, body, function (err) {
11                 if (err) throw err;
12                 var timestamp = new Date();
13                 console.log("Invoked first, happens second at %s", new Date());
14                 console.log("Wrote %s %s", outfile, timestamp);
15             });
16         }
17     });
18 }, 3000);
19 console.log("Invoked second, happens first at %s", new Date());
```

We can make this less messy as follows. It's still not as pretty as synchronous code, but is getting better. The main remaining issue is that `outfile` is still being imported from the enclosing scope into the local scopes of these functions.

```
1 #!/usr/bin/env node
2
3 var request = require('request');
4 var apiurl = 'http://nodejs.org/api/index.json';
5 var fs = require('fs');
6 var outfile = 'index4.json';
7
8 var cb_writefile = function (err) {
9     if (err) throw err;
10     var timestamp = new Date();
11     console.log("Invoked first, happens second at %s", new Date());
12     console.log("Wrote %s %s", outfile, timestamp);
```

```

13 };
14
15 var cb_parsebody = function(error, response, body) {
16     if (!error && response.statusCode == 200) {
17         fs.writeFile(outfile, body, cb_writefile);
18     }
19 };
20
21 request(apiurl, cb_parsebody);
22 console.log("Invoked second, happens first at %s", new Date());

```

We can fix the `outfile` issue as shown below. Note that we pass in the `apiurl` and `OUTFILE` variables to a new `make_request` function. Within the internal scope of `make_request`, we build up the functions that use `OUTFILE`. Finally, we capitalize `OUTFILE` to draw attention to it within the body of the function, similar to the way that one would capitalize a constant (as it's sort of serving as a constant within the scope of `make_request`). This isn't quite as nice as having so-called [referential transparency](#), as `cb_writefile` and `cb_parsebody` are still using an argument (namely `OUTFILE`) that isn't present in their explicit function signatures: `cb_writefile(err)` and `cb_parsebody(error, response, body)` respectively. So you can't tell what the true inputs to `cb_writefile` and `cb_parsebody` are simply by looking at these signatures, and this is how referential transparency is violated. That said, `make_request(apiurl, outfile)` does satisfy referential transparency and that would be the function that would be exposed if the code was built to be a library.

In general, this approach is a reasonable way to pass data through nested callbacks without monkeying around too much with function signatures. Most importantly, because we appropriately scoped `OUTFILE`, we have a much more general and easy to use function which is suitable for exposing to an external library. And we can now easily pass in another output file; see the last bit in the code.

```

1  #!/usr/bin/env node
2
3  var request = require('request');
4  var apiurl = 'http://nodejs.org/api/index.json';
5  var fs = require('fs');
6  var outfile = 'index-scoped.json';
7
8  var make_request = function(apiurl, OUTFILE) {
9
10     var cb_writefile = function (err) {
11         if (err) throw err;
12         var timestamp = new Date();
13         console.log("Wrote %s %s", OUTFILE, timestamp);
14     };
15
16     var cb_parsebody = function(error, response, body) {
17         if (!error && response.statusCode == 200) {
18             fs.writeFile(OUTFILE, body, cb_writefile);

```

```

19     }
20 };
21
22     request(apiurl, cb_parsebody);
23 };
24
25 make_request(apiurl, outfile);
26
27 // Because we appropriately scoped outfile,
28 // we have a much more general and easy to use function.
29 // We can now easily pass in another output file.
30 var another_outfile = 'index-another.json';
31 make_request(apiurl, another_outfile);

```

Now suppose that we want to nicely format the results of this API call. Below is one reasonably good way to do this.

```

1  #!/usr/bin/env node
2
3  var fs = require('fs');
4  var request = require('request');
5  var uu = require('underscore');
6  var apiurl = 'http://nodejs.org/api/index.json';
7  var outfile = 'index-parsed.json';
8
9  var module2obj = function(mod) {
10     var modregex = /\[([^\]]+)\]\[([^\]]+)\]/;
11     var match = modregex.exec(mod);
12     return {'name': match[1], 'file': match[2]};
13 };
14
15 var data2name_files = function(data) {
16     var notUndefined = function(xx) { return !uu.isUndefined(xx); };
17     var modules = uu.filter(uu.pluck(data.desc, 'text'), notUndefined);
18     return uu.map(modules, module2obj);
19 };
20
21 var body2json = function(body) {
22     return JSON.stringify(data2name_files(JSON.parse(body)), null, 2);
23 };
24
25 var make_request = function(apiurl, OUTFILE) {
26
27     var cb_writefile = function (err) {
28         if (err) throw err;
29         var timestamp = new Date();
30         console.log("Wrote %s %s", OUTFILE, timestamp);

```

```

31     };
32
33     var cb_parsebody = function(error, response, body) {
34         if (!error && response.statusCode == 200) {
35             fs.writeFile(OUTFILE, body2json(body), cb_writefile);
36         }
37     };
38
39     request(apiurl, cb_parsebody);
40 };
41
42 make_request(apiurl, outfile);
43
44 /* This code turns the raw output:
45
46     {
47         "source": "doc/api/index.markdown",
48         "desc": [
49             {
50                 "type": "list_start",
51                 "ordered": false
52             },
53             {
54                 "type": "list_item_start"
55             },
56             {
57                 "type": "text",
58                 "text": "[About these Docs](documentation.html)"
59             }
60         ]
61     }
62 Into something like this:
63
64     [
65         {
66             "name": "About these Docs",
67             "file": "documentation.html"
68         },
69         {
70             "name": "Synopsis",
71             "file": "synopsis.html"
72         },
73     ]
74 */

```

Note that to debug, we can write the raw JSON data pulled down by the first HTTP request to disk, and then load directly via `var data = require('./index.json');` or a sim-

ilar invocation. We can then play with this data in the REPL to build up a function like `data2name_files`. One of the issues with the callback style is that it makes working in the REPL more inconvenient, so tricks like this are useful.

Again, the main advantage of the asynchronous paradigm is that it allows us to begin the next part of the program without waiting for the current one to complete, which [can improve performance](#) and responsiveness for certain classes of problems. The main disadvantages are that async programming alone [will not](#) improve performance if we are compute- rather than IO-bound, asynchrony can make programs [harder](#) to reason about, and asynchronous code can lead to the ugly [nested callback](#) issue. One way around this is to use one of the node flow control libraries, like caolan's [async](#). Another is to wait for the [long-awaited](#) introduction of the `yield` keyword⁹ in node 0.11. To first order, once node 0.11 has a stable release, one should be able to replace `return` with `yield` and get the same conceptual and syntactical cleanliness of synchronous code, while preserving many of the useful features of asynchronous code.

Object-Oriented Programming (OOP), Prototypal Inheritance, and JS Objects

Now let's go through some of the basics of JS objects and the OOP style in JS. MDN as usual has a [good reference](#), and the [Eloquent JS](#) material is good as well; we'll complement and extend the definitions given therein.

Objects as dictionaries

The simplest way to work with objects is to think of them as similar to Python's dictionary type. We can use them to organize data that belongs together. Here's an example of a function that parses a URL scheme into its component pieces:

```
1 #!/usr/bin/env node
2 var parsedburl = function(dburl) {
3   var dbregex = /([~:]+):\\\/\\\/([~:~:]+):([~@]+)@([~:~:]+):([~d+])\\\/(.+)/;
4   var out = dbregex.exec(dburl);
5   return {'protocol': out[1],
6           'user': out[2],
7           'pass': out[3],
8           'host': out[4],
9           'port': out[5],
10          'dbpass': out[6]};
11 };
12 var pgurl = "postgres://myuser:mypass@example.com:5432/mydbpass";
13 console.log(parsedburl(pgurl));
14 /*
15 { protocol: 'postgres',
16   user: 'myuser',
```

⁹You can see this [post](#) and [this one](#). You'll want to invoke the sample code with `node --harmony` and also note that `gen.send()` may not yet be implemented. You can use `nvm install v0.11.5` to get the latest version of node and `nvm use v0.11.5` to switch to it for testing purposes, and then revert back with `nvm use v0.10.15`.

```

17     pass: 'mypass',
18     host: 'example.com',
19     port: '5432',
20     dbpass: 'mydbpass' }
21 */

```

You can take a look at Tim Caswell's series on JS Object Graphs ([1](#), [2](#), [3](#)) if you want to get into the guts of exactly what's going on when objects are instantiated in this way, and specifically how references between and within objects work.

Objects as dictionaries with functions

Without adding too much in the way of new concepts, we can use the special `this` keyword we described above to add a bit of intelligence to our objects. Now they aren't just dumb dictionaries anymore.

```

1  var dbobj = {
2      protocol: 'postgres',
3      user: 'myuser',
4      pass: 'mypass',
5      host: 'example.com',
6      port: '5432',
7      dbpass: 'mydbpass',
8      toString: function() {
9          return this.protocol + '://' +
10             this.user + ':' +
11             this.pass + '@' +
12             this.host + ':' +
13             this.port + '/' +
14             this.dbpass;
15      }
16  };
17
18  dbobj.toString();
19  // 'postgres://myuser:mypass@example.com:5432/mydbpass'

```

Here we equip the object with a function that pretty prints itself.

Objects as “classes”

Things start to get more complex when we start thinking about classes and inheritance. Unlike Java/C++/Python, JS is not a strictly object-oriented language. It actually has a prototypal inheritance model; see [here](#) for the difference. In practice, though, you can use it in much the same way that you'd create classes and class hierarchies in (say) Python.

The tricky part is that JS is flexible enough that you will see many different ways of setting up inheritance. Most of the time you'll want to either use simple objects-as-dictionaries (as in the previous section), or else use so-called [parasitic combination inheritance](#). Let's do an

example with Items for an e-commerce site like Amazon, building on some of the convenience functions defined by [Crockford](#), [Zakas](#), and [Bovell](#) (specifically the somewhat magical and now-standardized `Object.create` and the related `inheritPrototype`).

In our Amazon-ish example, each `Item` has a [Stock Keeping Unit](#) number (SKU) and a price, but other than that can vary widely and support different kinds of functions. A `Book` instance, for example, can search itself while a `Furniture` instance can calculate the floor area in square feet that they will occupy. Here's how we'd implement that; remember to use the `new` keyword when instantiating instances ([1](#), [2](#), [3](#)).

```
1  // Example of inheritance
2  //
3  // Item - sku, price
4  // Book - title, text, search
5  // Furniture - name, width, length
6
7
8  // 0. Preliminaries: this helper function copies the properties and methods
9  // of the parentObject to the childObject using the copyOfParent
10 // intermediary.
11 //
12 // See Zakas' writeup at goo.gl/o1wRG0 for details.
13 function inheritPrototype(childObject, parentObject) {
14     var copyOfParent = Object.create(parentObject.prototype);
15     copyOfParent.constructor = childObject;
16     childObject.prototype = copyOfParent;
17 }
18
19 // 1. Define the parent class constructor and add prototype methods
20 function Item(sku, price) {
21     this.sku = sku;
22     this.price = price;
23 }
24 Item.prototype.toString = function () {
25     return "SKU: " + this.sku + " | Price: " + this.price + " USD";
26 };
27
28 // 2. Define the subclass constructor, copy over properties and methods of
29 // Item, and then define a new function.
30 function Book(sku, price, title, text) {
31     Item.call(this, sku, price);
32     this.title = title;
33     this.text = text;
34 };
35
36 inheritPrototype(Book, Item);
37
38 Book.prototype.search = function(regexstr) {
```



```

39     var regex = RegExp(regexstr);
40     var match = regex.exec(this.text);
41     var out = '';
42     if(match !== null) {
43         var start = match.index;
44         var end = match.index + match[0].length;
45         var dx = 3;
46         var padstart = start - dx > 0 ? start - dx : start;
47         var padend = end + dx > 0 ? end + dx : end;
48         out = '...' + this.text.slice(padstart, padend) + '...';
49     }
50     return out;
51 };
52
53 // 3. Do one more subclass for illustrative purposes
54 function Furniture(sku, price, name, width, length) {
55     Item.call(this, sku, price);
56     this.name = name;
57     this.width = width;
58     this.length = length;
59 };
60
61 inheritPrototype(Furniture, Item);
62
63 Furniture.prototype.floorArea = function() {
64     return this.width * this.length;
65 };
66
67
68 // 4. Now let's test things out.
69 // Remember to use new!
70 var foo = new Item("ID:8908308", 43.27);
71 foo.sku;           // 'ID:8908308'
72 foo.price;         // 43.27
73 foo.toString();    // 'SKU: ID:8908308 | Price: 43.27 USD'
74
75 var bible = new Book("ID:123456", 101.02, "The Bible", "In the beginning there was");
76 bible.sku;         // 'ID:123456'
77 bible.price;       // 101.02
78 bible.toString();  // 'SKU: ID:123456 | Price: 101.02 USD'
79 bible.search('there'); // '...ng there wa...'
80 bible.search('therex'); // ''
81
82 var chair = new Furniture("ID:79808", 2020.32, "A chair", .5, 4.2);
83 chair.sku;         // 'ID:79808'
84 chair.price;       // 2020.32
85 chair.toString();  // 'SKU: ID:79808 | Price: 2020.32 USD'

```

```
86 | chair.floorArea(); // 2.1
```

This is a common way to do inheritance in JS. However, the syntax here is a bit kludgy and redundant, and a bit difficult to remember because JS doesn't have one true way of doing inheritance; see also the discussion in [Eloquent JS](#). You might want to build a small library for yourself to factor out the repeated code if you find yourself doing a lot of this; see [this post](#) and [pd](#). But beware: inheritance is not always the solution to everything. Often you can get away with something simple, like a type field and a [switch](#) statement, as shown:

```
1  // Inheritance isn't always the answer
2  //
3  // Instead of subclassing Honda, Mercedes, etc., we can often get away with
4  // switch statements or the like. rather than going with a class as an
5  // extremely heavyweight if/then statement.
6  //
7  // NOTE: in this particular example, we can actually replace the switch
8  // statement with an object lookup, but in general one might have more
9  // sophisticated logic in each switch case.
10
11 function Car(make, model) {
12     this.make = make;
13     this.model = model;
14 }
15 Car.prototype.toString = function () {
16     return "Make: " + this.make + " | Model: " + this.model;
17 };
18 Car.prototype.serviceIntervals = function() {
19     switch (this.make) {
20         case 'Honda':
21             out = [10000, 20000, 30000];
22             break;
23         case 'Mercedes':
24             out = [20000, 40000, 60000];
25             break;
26         default:
27             out = [5000, 10000, 20000];
28             break;
29     }
30     return out;
31 };
32
33 // Now let's instantiate some variables.
34
35 var accord = new Car('Honda', 'Accord');
36 var merc = new Car('Mercedes', 'S-Class');
37 var misc = new Car('Chrysler', 'Eminem Ride');
38
```

```

39 accord.toString();           // 'Make: Honda | Model: Accord'
40 merc.toString();             // 'Make: Mercedes | Model: S-Class'
41 misc.toString();             // 'Make: Chrysler | Model: Eminem Ride'
42 accord.serviceIntervals();   // [ 10000, 20000, 30000 ]
43 merc.serviceIntervals();     // [ 20000, 40000, 60000 ]
44 misc.serviceIntervals();     // [ 5000, 10000, 20000 ]

```

The lesson then is not to use inheritance when a switch statement, or an if/then will do instead. Here is another example where we use composition (**has_a**) rather than inheritance (**is_a**) to represent relationships between objects.

```

1  // Inheritance isn't always the answer, pt. 2
2  //
3  // Let's do another example where we show two different kinds of objects
4  // relate via composition.
5  function Wheel(isfront, isright) {
6      this.isfront = isfront;
7      this.isright = isright;
8  }
9  Wheel.prototype.toString = function () {
10     var pos1 = this.isfront ? 'f' : 'b'; // front/back
11     var pos2 = this.isright ? 'r' : 'l'; // right/left
12     return pos1 + pos2;
13 };
14 var wheel1 = new Wheel(true, true);
15 var wheel2 = new Wheel(true, false);
16 var wheel3 = new Wheel(false, true);
17 var wheel4 = new Wheel(false, false);
18
19 wheel1.toString(); // 'fr'
20 wheel2.toString(); // 'fl'
21 wheel3.toString(); // 'br'
22 wheel4.toString(); // 'bl'
23
24 // We use the invoke method in underscore.
25 // See underscorejs.org/#invoke
26 var uu = require('underscore');
27 function Car(make, model, wheels) {
28     this.make = make;
29     this.model = model;
30     this.wheels = wheels;
31 }
32 Car.prototype.toString = function () {
33     var jsdata = {'make': this.make,
34                  'model': this.model,
35                  'wheels': uu.invoke(this.wheels, 'toString')};
36     var spacing = 2;

```

```

37     return JSON.stringify(jsdata, null, spacing);
38 };
39
40 var civic = new Car('Honda', 'Civic', [wheel1, wheel2, wheel3, wheel4]);
41
42 /*
43 { make: 'Honda',
44   model: 'Civic',
45   wheels:
46     [ { isfront: true, isright: true },
47       { isfront: true, isright: false },
48       { isfront: false, isright: true },
49       { isfront: false, isright: false } ] }
50 */
51
52 console.log(civic.toString());
53 /*
54 {
55   "make": "Honda",
56   "model": "Civic",
57   "wheels": [
58     "fr",
59     "fl",
60     "br",
61     "bl"
62   ]
63 }
64 */

```

Note that while you should know how to work with arrays and dictionaries by heart, it's ok to look up the syntax for defining objects as you won't be writing new classes extremely frequently.

Heuristics for OOP in JS

Here are a few tips for working with OOP in JS.

- *Consider using switches and types rather than inheritance:* You usually don't want to use inheritance if you can get away with a simple type field and switch statement in a method. For example, you probably don't want to make `Honda` and `Mercedes` subclasses of `Car`. Instead you'd do something like `car.model = 'Mercedes'`; combined with a method that uses the `switch` statement on the `car.model` field.
- *Consider using composition rather than inheritance:* You don't want to make a `Wheel` a subclass of `Car`. Conceptually, a `Car` instance would have a list of four `Wheel` instances as a field, e.g. `car.wheels = [wheel1, wheel2, wheel2, wheel4]`. A relationship does exist, but it's a composition relationship (`Car` has_a `Wheel`) rather than an inheritance relationship (`Wheel` is_a `Car`).

- *Use shallow hierarchies if you use inheritance:* In the event that inheritance really is the right solution, where both functions and behavior change too much to make type fields/switches reasonable, then go ahead but use a shallow inheritance hierarchy. A very deep inheritance hierarchy is often a symptom of using a class definition like an if/else statement.
- *Be careful about mutual exclusivity with inheritance.* A seemingly reasonable example would be **User** as a superclass and **MerchantUser** and **CustomerUser** as subclasses in an online marketplace app. Both of these **User** subclasses would have email addresses, encrypted passwords, and the like. But they would also have enough different functionality (e.g. different login pages, dashboards, verification flows, etc) that you'd actually be benefiting from inheritance. However, an issue arises if your Merchant users want to also serve as Customers, or vice versa. For example, Airbnb hosts can book rooms, and people who book rooms can serve as hosts. Don't box yourself into a corner with inheritance; try to look for truly mutually exclusive subclasses.
- *Think about classes and instances in terms of tables and rows, respectively.* In a production webapp most of your classes/instances will be respectively tied to tables/rows in the database via the ORM, and inheritance hierarchies with ORMs tend to be fairly shallow. This is the so-called [Active Record](#) paradigm.
- *Make your objects talk to themselves:* If you find yourself writing a function that is picking apart the guts of an object and calculating things, you should consider moving that function into an object method. Note how our **dbobj** object had a **toString** method that used the **this** keyword to introspect and print itself. Grouping code with the data it operates on in this way helps to manage complexity.
- *Verbally articulate your flow. Nouns are classes, verbs are functions or methods.* A good way to define the classes and functions for your new webapp is to write down a few paragraphs on how the app works. Any recurrent nouns are good candidates for classes, and any recurring verbs are good candidates for functions or methods. For example, in a Bitcoin app you might talk about users with different Bitcoin addresses sending and receiving payments. In this case, **User** and **Address** are good classes, and **send** and **receive** are good functions.

To summarize, then, you should now have a handle on a common suite of JS concepts that apply to both frontend JS (in the browser) and server-side JS (in node). These concepts include the basic global objects in JS interpreters and how to use them, techniques for functional programming (FP) in JS, and techniques for working with prototypes and objects to do something very similar to object-oriented programming (OOP) in JS.