Startup Engineering
Balaji S. Srinivasan
Stanford University

Class: class.coursera.org/startup-001
Signup: coursera.org/course/startup
Forums: bit.ly/startup-eng-forum

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 languages[1] respectively).

[1]Though 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 prompt[2], 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 way[3] 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:

```
#!/usr/bin/env node
// http://projecteuler.net/problem=1
// If we list all the natural numbers below 10 that are multiples of 3 or 5,
// we get 3, 5, 6 and 9. The sum of these multiples is 23.
//
// Find the sum of all the multiples of 3 or 5 below 1000.

var divides = function(a, b) {
    return b % a === 0;
};

var anydivide = function(as, b) {
    for(var ii in as) {
        if(divides(as[ii], b)) {
            return true;
        }
    }
    return false;
};

var sum = function(arr) {
    var cc = 0;
```

[2]In 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.

[3]If 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.

```
    for(var ii in arr) {
        cc += arr[ii];
    }
    return cc;
};

var fizzbuzz = function(factors, max) {
    var out = [];
    for(var nn = 1; nn < max; nn += 1) {
        if(anydivide(factors, nn)) {
            out.push(nn);
        }
    }
    return sum(out);
};

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:

```
#!/usr/bin/env node
// http://projecteuler.net/problem=2
//
// Each new term in the Fibonacci sequence is generated by adding the
// previous two terms. By starting with 1 and 2, the first 10 terms will be:
//
// 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
//
// By considering the terms in the Fibonacci sequence whose values do not
// exceed four million, find the sum of the even-valued terms.

// Fibonacci: closed form expression
// wikipedia.org/wiki/Golden_ratio#Relationship_to_Fibonacci_sequence
var fibonacci = function(n) {
    var phi = (1 + Math.sqrt(5))/2;
    return Math.round((Math.pow(phi, n + 1) - Math.pow(1 - phi, n + 1))/Math.sqrt(5));
};

var iseven = function(n) {
    return n % 2 === 0;
};

var sum = function(arr) {
    var cc = 0;
    for(var ii in arr) {
        cc += arr[ii];
    }
    return cc;
};

var fibsum = function(max) {
    var value = 0;
    var ii = 1;
    var out = [];
    var flag = false;
    while(value < max) {
        value = fibonacci(ii);
        flag = iseven(value);
        ii += 1;
        if(flag && value < max) {
            out.push(value);
        }
    }
    return sum(out);
};

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.

```
// Important Array global methods

// length
var log = console.log;
var foo = [1, 2, 3, 3, 6, 2, 0];
log(foo.length); // 7

// concat: create new array
var bar = [4, 5, 6];
var baz = foo.concat(bar);

log(foo); // [ 1, 2, 3, 3, 6, 2, 0 ]
log(bar); // [ 4, 5, 6 ]
log(baz); // [ 1, 2, 3, 3, 6, 2, 0, 4, 5, 6 ]

// push/unshift to add to end/beginning
foo.push(10);
foo.unshift(99);
log(foo); // [ 99, 1, 2, 3, 3, 6, 2, 0, 10 ]

// pop/shift to remove the last/first element
var last = foo.pop();
var first = foo.shift();
log(last);  // 10
log(first); // 99
log(foo); // [ 1, 2, 3, 3, 6, 2, 0 ]

// join: combine elements in an array into a string
log('<tr><td>' + bar.join('</td><td>') + "</td></tr>");
// <tr><td>4</td><td>5</td><td>6</td></tr>

// slice: pull out a subarray
var ref = foo.slice(3, 6);
log(ref); // [ 3, 6, 2 ]
log(foo); // [ 1, 2, 3, 3, 6, 2, 0 ]
ref[0] = 999;
log(ref); // [ 999, 6, 2 ]
log(foo); // [ 1, 2, 3, 3, 6, 2, 0 ]

// reverse: MODIFIES IN PLACE
foo.reverse();
log(foo); // [ 0, 2, 6, 3, 3, 2, 1 ]
foo.reverse();
log(foo); // [ 1, 2, 3, 3, 6, 2, 0 ]

// sort: MODIFIES IN PLACE
foo.sort();
log(foo); // [ 0, 1, 2, 2, 3, 3, 6 ]

// inhomogeneous types work
var arr = [99, 'mystr', {'asdf': 'alpha'}];

// JS will do the best it can if you invoke default methods.
// So make sure to use the REPL if you're doing something exotic.
arr.sort();
log(arr); // [ 99, { asdf: 'alpha' }, 'mystr' ]
arr.join(','); // '99,[object Object],mystr'


// Be very careful in particular when working with inhomogeneous arrays that
// contain objects. Methods like slice will not do deep copies, so modifications
// of the new array can propagate back to the old one when dealing with objects.
var newarr = arr.slice(0, 2);
console.log(newarr);      // [ 99, { asdf: 'alpha' } ]
console.log(arr);         // [ 99, { asdf: 'alpha' }, 'mystr' ]
newarr[1].asdf = 'ooga';
console.log(newarr);      // [ 99, { asdf: 'ooga' } ]
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 relative[4] 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. Constructing Date instances.
//
//     Make sure to use "new Date", not "Date" when instantiating.
//
var year = 2000;
var month = 02;
var day = 29;
var hour = 12;
var minute = 30;
var second = 15;
var millisecond = 93;
var milliseconds_since_jan_1_1970 = 86400 * 10 * 1000;
var iso_timestamp = '2003-05-23T17:00:00Z';

var dt0 = Date(); // BAD - just a string, not a Date instance
var dt1 = new Date(); // GOOD - a real Date instance using new
var dt2 = new Date(milliseconds_since_jan_1_1970);
var dt3 = new Date(iso_timestamp);
```

[4]The problem with this convention is that after 2147483647 millseconds, 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.

```
var dt4 = new Date(year, month, day);
var dt5 = new Date(year, month, day, hour, minute, second, millisecond);

/*
> dt1
Mon Aug 05 2013 07:19:45 GMT-0700 (PDT)
> dt2
Sat Jan 10 1970 16:00:00 GMT-0800 (PST)
> dt3
Fri May 23 2003 10:00:00 GMT-0700 (PDT)
> dt4
Wed Mar 29 2000 00:00:00 GMT-0800 (PDT)
> dt5
Wed Mar 29 2000 12:30:15 GMT-0800 (PDT)
*/

// 2. Date classmethods - these return milliseconds, not Date instances.

// now: gives number of milliseconds since Jan 1, 1970 00:00 UTC
var milliseconds_per_year = (86400 * 1000 * 365);
var years_from_epoch = function(ms) {
    console.log(Math.floor(ms/milliseconds_per_year));
};
years_from_epoch(Date.now()); // 43

// parse: takes in ISO timestamp and returns milliseconds since epoch
years_from_epoch(Date.parse('2003-05-23T17:00:00Z')); // 33

// UTC: Constructor that returns milliseconds since epoch
years_from_epoch(Date.UTC(year, month, day, hour, minute,
                          second, millisecond)); // 30


// 3. Date Examples

// 3.1 - Calculating the difference between two dates
console.log(dt3); // Fri May 23 2003 10:00:00 GMT-0700 (PDT)
console.log(dt4); // Wed Mar 29 2000 00:00:00 GMT-0800 (PDT)
var ddt = dt3 - dt4;
var ddt2 = dt3.getTime() - dt4.getTime();
console.log(ddt);  // 99392400000
console.log(ddt2); // 99392400000
console.log(ddt / milliseconds_per_year); // 3.151712328767123

// 3.2 - Get JSON data, parsing strings into Date instances,
//       and then return docs structure.
var data2docs = function(data) {
    var docs = [];
    var offset = 'T12:00:00-05:00'; // Explicitly specify time/timezone.
    var dt, ii, dtnew;
    for(ii = 0; ii < data.results.length; ii += 1) {
        dt = data.results[ii].publication_date; // looks like: '2012-12-14'
        dti = new Date(dt + offset);
        docs.push({'title':data.results[ii].title, 'publication_date': dti});
    }
    return docs;
};

var docs2console = function(docs) {
    for(var ii = 0; ii < docs.length; ii++) {
        doc = docs[ii];
        console.log('Date: %s\nTitle: %s\n', doc.publication_date, doc.title);
    }
};

var apiurl = "https://www.federalregister.gov/api/v1/articles/" +
"03-12969,2012-30312,E8-24781.json?fields%5B%5D=title" +
"&fields%5B%5D=publication_date";
var request = require('request'); // npm install request
var printdata = function(apiurl) {
    request(apiurl, function (error, response, body) {
        if (!error && response.statusCode == 200) {
            data = JSON.parse(body);
            docs2console(data2docs(data));
        }
    });
};

/* This produces the following output:
Date: Fri May 23 2003 10:00:00 GMT-0700 (PDT)
Title: National Security Agency/Central Security Service (NSA/CSS) Freedom of Information Ac

Date: Fri Dec 14 2012 09:00:00 GMT-0800 (PST)
Title: Human Rights Day and Human Rights Week, 2012

Date: Mon Oct 20 2008 10:00:00 GMT-0700 (PDT)
Title: Tarp Capital Purchase Program
*/
```

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 examples[5] of its use:

```
// Regexp examples

// 1. Test for presence, identify location
var ex1 = "The quick brown fox jumped over the lazy dogs.";
var re1 = /(cr|l)azy/;
console.log(re1.test(ex1));
// true
console.log(re1.exec(ex1));
/*
[ 'lazy',
  'l',
  index: 36,
  input: 'The quick brown fox jumped over the lazy dogs.' ]
*/


// 2. Global matches
var ex2 = "Alpha Beta Gamma Epsilon Omicron Theta Phi";
var re2 = /(\w+a )/;
var re2g = /(\w+a )/g;

re2.exec(ex2);
re2.exec(ex2);
/* Without the /g, repeating the match doesn't do anything here.

[ 'Alpha ',
  'Alpha ',
```

[5]Note 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.

```
  index: 0,
  input: 'Alpha Beta Gamma Epsilon Omicron Theta Phi' ]

[ 'Alpha ',
  'Alpha ',
  index: 0,
  input: 'Alpha Beta Gamma Epsilon Omicron Theta Phi' ]
*/

re2g.exec(ex2);
re2g.exec(ex2);
re2g.exec(ex2);
re2g.exec(ex2);
re2g.exec(ex2);
/* With the /g, repeating the match iterates through all matches.

[ 'Alpha ',
  'Alpha ',
  index: 0,
  input: 'Alpha Beta Gamma Epsilon Omicron Theta Phi' ]

[ 'Beta ',
  'Beta ',
  index: 6,
  input: 'Alpha Beta Gamma Epsilon Omicron Theta Phi' ]

[ 'Gamma ',
  'Gamma ',
  index: 11,
  input: 'Alpha Beta Gamma Epsilon Omicron Theta Phi' ]

[ 'Theta ',
  'Theta ',
  index: 33,
  input: 'Alpha Beta Gamma Epsilon Omicron Theta Phi' ]

null
*/

// We can formalize the process of iterating through the
// matches till we hit a null with the following function:
var allmatches = function(rex, str) {
    var matches = [];
    var match;
    while(true) {
        match = rex.exec(str);
        if(match !== null) { matches.push(match); }
        else { break; }
    }
    return matches;
};
allmatches(re2g, ex2);
/*
[ [ 'Alpha ',
    'Alpha ',
    index: 0,
    input: 'Alpha Beta Gamma Epsilon Omicron Theta Phi' ],
  [ 'Beta ',
    'Beta ',
    index: 6,
    input: 'Alpha Beta Gamma Epsilon Omicron Theta Phi' ],
  [ 'Gamma ',
    'Gamma ',
    index: 11,
    input: 'Alpha Beta Gamma Epsilon Omicron Theta Phi' ],
  [ 'Theta ',
    'Theta ',
    index: 33,
    input: 'Alpha Beta Gamma Epsilon Omicron Theta Phi' ] ]
*/


// 3. Case-insensitive
var ex3 = "John is here.";
var re3s = /JOHN/;
var re3i = /JOHN/i;
re3s.test(ex3); // false
re3i.test(ex3); // true


// 4. Multiline
var ex4 = "Alpha beta gamma.\nAll the king's men.\nGermany France Italy.";
var re4s = /^G/;
var re4m = /^G/m;
re4s.test(ex4); // false
re4m.test(ex4); // true


// 5. Parsing postgres URLs
var pgurl = "postgres://myuser:mypass@example.com:5432/mydbpass";
var pgregex = /postgres:\/\/([^:]+):([^@]+)@([^:]+):(\d+)\/(.+)/;
var flag = pgregex.test(pgurl); // true
var out = pgregex.exec(pgurl);
/*
```

```
[ 'postgres://myuser:mypass@example.com:5432/mydbpass',
  'myuser',
  'mypass',
  'example.com',
  '5432',
  'mydbpass',
  index: 0,
  input: 'postgres://myuser:mypass@example.com:5432/mydbpass' ]
*/


// 6. Parsing postgres URLs in a function (more advanced)
//    Here, we use the object and zip methods from underscore to organize
//    the regex-parsed fields in a nice, easy-to-use data structure.
var uu = require('underscore');
var parsedburl = function(dburl) {
    var dbregex = /([^:]+):\/\/([^:]+):([^@]+)@([^:]+):(\d+)\/(.+)/;
    var out = dbregex.exec(dburl);
    var fields = ['protocol', 'user', 'pass', 'host', 'port', 'dbpass'];
    return uu.object(uu.zip(fields, out.slice(1, out.length)));
};

console.log(parsedburl(pgurl));
/*
{ protocol: 'postgres',
  user: 'myuser',
  pass: 'mypass',
  host: 'example.com',
  port: '5432',
  dbpass: 'mydbpass' }
*/
```

In general, it's more convenient to use the forward-slash syntax than to actually write out `RegExp`. Note that if you are heavily using regexes for parsing files, you may want to write a formal parser instead, use a csv or xml library, or make use of the built-in JSON parser.

Math

The Math object holds some methods and constants for basic precalculus. Perhaps the most useful are `floor`, `ceil`, `pow`, and `random`.

```
// Math examples

// 1. Enumerating the available Math methods
//    Object.getOwnPropertyNames allows you to do something like Python's dir.
var log = console.log;
log(Object.getOwnPropertyNames(Math));
```

```

/*
[ 'E',
  'LN10',
  'LN2',
  'LOG2E',
  'LOG10E',
  'PI',
  'SQRT1_2',
  'SQRT2',
  'random',
  'abs',
  'acos',
  'asin',
  'atan',
  'ceil',
  'cos',
  'exp',
  'floor',
  'log',
  'round',
  'sin',
  'sqrt',
  'tan',
  'atan2',
  'pow',
  'max',
  'min' ]
*/


// 2. Generating random integers between a and b, not including the upper bound.
var randint = function(a, b) {
    var frac = Math.random();
    return Math.floor((b-a)*frac + a);
};


// 3. Recapitulating constants
Math.pow(Math.E, Math.LN2);  // 1.9999999999999998
Math.pow(Math.E, Math.LN10); // 10.000000000000002
Math.pow(10, Math.LOG10E) - Math.E;  // 0
Math.pow(2, Math.LOG2E) - Math.E; // 0


// 4. Determining the effective growth rate from the start time (in months),
// the stop time (in months), the initial user base, and the final user
// base.
//
// init * 2^{(stop-start)/tau} = fin
//
// tau = (stop-start)/log2(fin/init)
//
var log2 = function(xx) {
    return Math.log(xx)/Math.LN2;
};

var doublingtime = function(start, stop, init, fin) {
    var dt = (stop-start);
    var fold = fin/init;
    return dt/log2(fold);
};

log(doublingtime(0, 10, 1, 16)); // 2.5
var tau = doublingtime(0, 24, 1, 9); // 7.571157042857489
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 math[6] 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:

```
// String examples

// 1. Treat String as Array of characters
var log = console.log;
var sx1 = "The quick brown fox jumped over the lazy dogs.";
sx1.charAt(10); // 'b'
sx1.slice(10, 13); // 'bro'
sx1.substring(10, 13); // 'bro'
sx1.substr(10, 13); // 'brown fox jum'
```

[6]If you are interested in rendering math with JS, look at the powerful MathJax library for generating LaTeX 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).

```
sx1.length; // 46

// 2. Compare strings using alphabetical ordering
var sx2 = "alpha";
var sx3 = "beta";
log(sx2 < sx3); // true

// 3. Replace substrings via string or regex
var sx4 = sx2.replace('ph', 'foo');
log(sx2); // 'alpha'
log(sx4); // 'alfooa'

var sx5 = sx2.replace(/a$/, 'bar'); // NOTE regex
log(sx2); // 'alpha'
log(sx4); // 'alphbar'

// 4. Change case (lower, upper)
log(sx2.toUpperCase()); // 'ALPHA'

// 5. Trim strings
var sx6 = " " + ['Field1', 'Field2', 'Field3'].join("\t") + "\n";
var sx7 = sx6.trimRight();
var sx8 = sx6.trimLeft();
var sx9 = sx6.trim();
log(sx6);  // ' Field1\tField2\tField3\n'
log(sx7);  // ' Field1\tField2\tField3'
log(sx8);  // 'Field1\tField2\tField3\n'
log(sx9);  // 'Field1\tField2\tField3'

// 6. Split strings (useful for simple parsing)
var fields = sx9.split("\t");
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 subset[7] 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.

[7]If 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.

```
// JSON examples

// 1. List methods on JSON object
var log = console.log;
Object.getOwnPropertyNames(JSON) // [ 'parse', 'stringify' ]

// 2. Using JSON.parse to deserialize JSON strings.
//
//    Note that you use double quotes within JSON and single quotes to
//    encapsulate the entire string
var jsdata = '[ {"asdf":9, "bar":10}, 18, "baz"]';
var data = JSON.parse(jsdata);
log(data[0].asdf); // 9

// You can also do this with eval. But don't do that. Use JSON.parse instead.
var data2 = eval(jsdata);
log(data2[0].asdf); // 9

// 3. Using JSON.stringify to serialize JS objects.
//
//    While strings, numbers, arrays, and dictionaries/objects are generally
//    safe, note that Regexp instances don't have a good default
//    serialization and thus need special handling.
var dt = new Date('2003-05-23T17:00:00Z');
var rex = /(cr|l)/;
var data3 = [9, {"foo": dt, "bar": rex, "baz": {"quux": "a", "alpha": [77, 3]}}, 11];
log(data3);
/*
[ 9,
  { foo: Fri May 23 2003 10:00:00 GMT-0700 (PDT),
    bar: /(cr|l)/,
    baz: { quux: 'a', alpha: [Object] } },
  11 ]
*/

// Note that the Regexp instance is serialized to {} rather than "/(cr|l)/"
var data3str = JSON.stringify(data3);
// '[9,{"foo":"2003-05-23T17:00:00.000Z","bar":{},"baz":{"quux":"a","alpha":[77,3]}},11]'

// We can restore the data structure as shown. Note again the the restoration
// is only as good as the serialization. Make sure to look at the raw JSON string
// output
var data4 = JSON.parse(data3str);
log(data4);
/*
[ 9,
  { foo: '2003-05-23T17:00:00.000Z',
    bar: {},
    baz: { quux: 'a', alpha: [Object] } },
  11 ]
*/
```

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.

```
// Error examples

// 1. List all methods in Error
Object.getOwnPropertyNames(Error);
/*
[ 'length',
  'name',
  'arguments',
  'caller',
  'prototype',
  'captureStackTrace',
  'stackTraceLimit' ]
*/

// 2. An example of try/catch
var log = console.log;
var div = function(a, b) {
    try {
        if(b === 0) {
            throw new Error("Divided by Zero");
        } else {
            return a/b;
        }
    } catch(e) {
        log('name\n%s\n\nmessage\n%s\n\nstack\n%s', e.name, e.messsage, e.stack);
    }
};

/*
name
Error
```

```

message
undefined

stack
Error: Divided by Zero
    at div (repl:1:79)
    at repl:1:2
    at REPLServer.self.eval (repl.js:112:21)
    at Interface.<anonymous> (repl.js:239:12)
    at Interface.EventEmitter.emit (events.js:95:17)
    at Interface._onLine (readline.js:202:10)
    at Interface._line (readline.js:531:8)
    at Interface._ttyWrite (readline.js:767:16)
    at ReadStream.onkeypress (readline.js:99:10)
    at ReadStream.EventEmitter.emit (events.js:98:17)
*/


// 3. Returning an Error object directly
var div2 = function(a, b) {
    if(b === 0) {
        return new Error("Divided by Zero");
    } else {
        return a/b;
    }
};
var err = div2(4, 0);
log(err.stack);
/*
Error: Divided by Zero
    at div2 (repl:1:62)
    at repl:1:11
    at REPLServer.self.eval (repl.js:112:21)
    at repl.js:249:20
    at REPLServer.self.eval (repl.js:122:7)
    at Interface.<anonymous> (repl.js:239:12)
    at Interface.EventEmitter.emit (events.js:95:17)
    at Interface._onLine (readline.js:202:10)
    at Interface._line (readline.js:531:8)
    at Interface._ttyWrite (readline.js:767:16)
*/


// 4. Using custom error types.
//
//  Modified from Zip Code example here:
```

```
//  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/throw#Examp
//
//  Note that for a real email parser, you will want to use an existing
//  regex rather than writing your own, as email formats can get
//  surprisingly complex. See RFC 2822 and here:
//
//  http://www.regular-expressions.info/email.html
//  http://tools.ietf.org/html/rfc2822#section-3.4.1
//
//  Better yet, if you really want to diagnose problems with invalid email
//  addresses use a full function that returns specific errors for different
//  cases. See here: stackoverflow.com/q/997078

var Email = function(emstr) {
    var regex = /([^@]+)@([^\.]+)\.([^\.]+)/;
    if (regex.test(emstr)) {
        var match = regex.exec(emstr);
        this.user = match[1];
        this.domain = match[2];
        this.tld = match[3];
        this.valueOf = function() {
            return this.value;
        };
        this.toString = function() {
            return this.user + '@' + this.domain + '.' + this.tld;
        };
   } else {
      throw new EmailFormatException(emstr);
   }
};

var EmailFormatException = function(value) {
   this.value = value;
   this.message = "not in a@b.c form";
   this.toString = function() {
      return this.value + this.message;
   };
};

var EMAIL_INVALID = -1;
var EMAIL_UNKNOWN = -1;

var parseEmail = function(instr) {
   try {
      em = new Email(instr);
   } catch (e) {
      if (e instanceof EmailFormatException) {
            return EMAIL_INVALID;
        } else {
            return EMAIL_UNKNOWN_ERROR;
        }
    }
    return em;
};

// Make sure to include new when using the Email constructor directly
var foo = Email('john@gmail.com'); // Doesn't work
var bar = new Email('joe@gmail.com'); // Works

// Here's what happens when we invoke parseEmail
parseEmail('john@gmailcom'); // -1 == EMAIL_INVALID
parseEmail('johngmail.com'); // -1 == EMAIL_INVALID
parseEmail('john@gmail.com');
/*
{ user: 'john',
  domain: 'gmail',
  tld: 'com',
  valueOf: [Function],
  toString: [Function] }
*/
```

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.

```
// Built-in examples

/*
  1. decodeURI and encodeURI: escape special characters in URIs

  These two functions ensure that special characters like brackets and
  slashes are escaped in URI strings. Let's illustrate with an
  example API call to the federalregister.gov site.
*/
var apiurl = "https://www.federalregister.gov/api/v1/articles/" +
"03-12969,2012-30312,E8-24781.json?fields%5B%5D=title" +
"&fields%5B%5D=publication_date";
```

```
decodeURI(apiurl);
// 'https://www.federalregister.gov/api/v1/articles/03-12969,
// 2012-30312,E8-24781.json?fields[]=title&fields[]=publication_date'
encodeURI(decodeURI(apiurl));
// 'https://www.federalregister.gov/api/v1/articles/03-12969,
// 2012-30312,E8-24781.json?fields%5B%5D=title&fields%5B%5D=publication_date'

/*
   2. typeof: gives the type of a JS variable.

   Note: typeof cannot be used by itself, as it won't distinguish between
   many kinds of objects (see here: http://stackoverflow.com/a/7086473).  It
   is, however, used by many libraries in conjunction with instanceof and
   various kinds of duck-typing checks. See here:
   tobyho.com/2011/01/28/checking-types-in-javascript

   In general, metaprogramming in JS without a reliable library (something
   like modernizr.js, but for types rather than browser features) is a pain
   if you want to do it in full generality; it's not as regular as Python.
*/

typeof 19
// 'number'
typeof "foo"
// 'string'
typeof {}
// 'object'

/*
   3. parseInt and parseFloat: convert strings to ints and floats

   Note that parseInt takes a base as the second argument, and can be used
   to convert up to base 36.
*/

parseInt('80', 10) // = 8*10^1 + 0*10^0
// 80
parseInt('80', 16) // = 8*16^1 + 0*16^0
// 128
parseFloat('89803.983')
// 89803.983

/*
   4. Object.getOwnPropertyNames

   Useful routine that can remind you which methods exist on a particular
   object.
```

```
*/

Object.getOwnPropertyNames(JSON)
// [ 'parse', 'stringify' ]


/*
   5. Object.getOwnPropertyDescriptor

   Use this to introspect the fields of a given object or module.
*/

var uu = require('underscore');
Object.getOwnPropertyDescriptor(uu, 'map');
/*
{ value: [Function],
  writable: true,
  enumerable: true,
  configurable: true }
*/
```

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:

```
#!/usr/bin/env node

// Implementation 1: Procedural
var log = console.log;
var n = 5;
var out = [];
for(var i = 0; i < n; i++) {
    out.push(i * i);
}
log(out); // [ 0, 1, 4, 9, 16 ]
```

```
// Implementation 2: Functional Programming
// Functions as first class variables
//
// Allows us to abstract out the loop and the function applied within, such
// that we can swap in a new function or a new kind of iteration.
var sq = function(x) { return x * x;};
var cub = function(x) { return x * x * x;};
var loop = function(n, fn) {
    var out = [];
    for(var i = 0; i < n; i++) {
        out.push(fn(i));
    }
    return out;
};
var loopeven = function(n, fn) {
    var out = [];
    for(var i = 0; i < n; i++) {
        if(i % 2 === 0) {
            out.push(fn(i));
        } else {
            out.push(i);
        }
    }
    return out;
};
log(loop(n, sq));      // [ 0, 1, 4, 9, 16 ]
log(loop(n, cub));     // [ 0, 1, 8, 27, 64 ]
log(loopeven(n, sq));  // [ 0, 1, 4, 3, 16 ]
log(loopeven(n, cub)); // [ 0, 1, 8, 3, 64 ]


// Implementation 3: Functional Programming with underscorejs.org
// Note the use of the map and range methods.
//  - range: generate an array of numbers between 0 and n
//  - map: apply a function to a specified array
var uu = require('underscore');
log(uu.map(uu.range(0, n), sq)); // [ 0, 1, 4, 9, 16 ]
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:

```
#!/usr/bin/env node
// http://projecteuler.net/problem=1
// If we list all the natural numbers below 10 that are multiples of 3 or 5,
// we get 3, 5, 6 and 9. The sum of these multiples is 23.
//
// Find the sum of all the multiples of 3 or 5 below 1000.
//
// We'll make heavy use of underscore here to illustrate some FP techniques.
var uu = require('underscore');

// For the anydivide function
//  - uu.map takes an array and a univariate function, and applies the
//    function element-wise to the array, returning a new array.
//
//  - We define divfn via a closure, so that it takes one variable and
//    returns one output. This can then be passed to uu.map.
//
//  - uu.any takes an array of booleans and returns true if and only if one
//    of them is true, and false otherwise.
var anydivide = function(as, b) {
    var divfn = function(a) { return b % a === 0; };
    return uu.any(uu.map(as, divfn));
};

// For the sum function
//
//  - uu.reduce takes an array and a function, and applies that function
//    iteratively to each element of the array, starting from the first element.
//  - The anonymous function we pass in as the second argument just adds adjacent
//    elements together.
//
var sum = function(arr) {
    return uu.reduce(arr, function(a, b) { return a + b;});
};

// For the fizzbuzz function
//
//  - We define divfn as a closure, using the factors variable. Again, this
//    is so that it takes one argument, returns one output, and can be passed to map.
//
//  - We use uu.filter to iterate over an array and return all elements
//    which pass a condition, specified by a function that returns booleans.
//    The array is the first argument to uu.filter and the boolean-valued
//    func is the second argument.
//
//  - We use uu.range to generate an array of numbers between two specified
//  - values.
//
var fizzbuzz = function(factors, max) {
    var divfn = function(nn) { return anydivide(factors, nn); };
    var divisible = uu.filter(uu.range(1, max), divfn);
    return sum(divisible);
};

// console.log(fizzbuzz([3, 5], 10));
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 often[8] 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. Partial function application in JS
//
// The concept of partial function application (aka currying, named after
// Haskell Curry the mathematician).
var sq = function(x) { return x * x;};
var loop = function(n, fn) {
    var out = [];
    for(var i = 0; i < n; i++) {
        out.push(fn(i));
    }
    return out;
};

```

[8]Note 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.

```
// We can use the 'bind' method on Functions to do partial evaluation
// of functions, as shown.
//
// Here, loopN now accepts only one argument rather than two.  (We'll cover
// the null argument passed in to bind when we discuss 'this').
loop(10, sq); // [ 0, 1, 4, 9, 16, 25, 36, 49, 64, 81 ]
var loopN = loop.bind(null, 10); // n = 10
loopN(sq); // [ 0, 1, 4, 9, 16, 25, 36, 49, 64, 81 ]

// 2. Implementing currying in underscore is easy with the 'partial' method.
var uu = require('underscore');
loop6 = uu.partial(loop, 6);
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:

```
// Scope in JS

// 1. Unintentional global variables
//
//    This can occur if you forget to declare cc within foo.
var cc = 99;
var foo = function(aa) {
    var bb = 7;
    console.log("aa = " + aa);
    console.log("bb = " + bb);
    console.log("cc = " + cc);
};
foo(22);
/*
aa = 22
bb = 7
cc = 99
*/

// 2. Intentional global variables
//
//    This is the recommended syntax within node if you really
//    want to use globals. You can accomplish the same thing with
//    window in the browser.
var dd = 33;
var bar = function(aa) {
    var bb = 7;
    console.log("aa = " + aa);
    console.log("bb = " + bb);
    console.log("dd = " + global.dd);
};
bar(22);
/*
aa = 22
bb = 7
dd = 33
*/


// 3. Scope in statically defined functions
//
//    The aa defined within this functions' argument and the bb within its
//    body override the external aa and bb. This function is defined
//    'statically', i.e. it is not generated on the fly from input arguments
//    while the program is running.
var aa = 7;
var bb = 8;
var baz = function(aa) {
    var bb = 9;
    console.log("aa = " + aa);
    console.log("bb = " + bb);
};
baz(16);
/*
aa = 16
bb = 9
*/
```

```javascript


// 4. Scope in dynamically defined functions (simple)
//
//    Often we'll want to build a function on the fly, in which case the use
//    of variables from the enclosing scope is a feature rather than a
//    bug. This kind of function is called a closure. Let's first do a
//    trivial example.
//
//    Here, buildfn is intentionally using the increment variable
//    passed in as an argument in the body of the newfn, which is
//    being created on the fly.
var buildfn = function(increment) {
    var newfn = function adder(xx) { return xx + increment; };
    return newfn;
};
buildfn(10); // [Function: adder]
buildfn(10)(17); // 27
var add3 = buildfn(3);
var add5 = buildfn(5);
add3(add5(17)); // 25


//    Here's a more complex example where we pass in a function rather
//    than simply assuming buildfn2 will be doing addition.
var buildfn2 = function(yy, binaryoperator) {
    var cc = 10;
    var newfn = function binop(xx) { return cc*binaryoperator(xx, yy); };
    return newfn;
};
var pow = function(aa, bb) { return Math.pow(aa, bb);}
var fn = buildfn2(3, pow);
fn(5); // 10 * Math.pow(5, 3)


// 5. Scope in dynamically defined functions (more advanced)
//
//    Here's a more realistic example, where we define a meta-function that
//    takes two other functions and snaps them together to achieve an
//    effect.
var compose = function(f, g) {
    var h = function(x) {
        return f(g(x));
    };
    return h;
};

```

```
var jsdata = '[ {"asdf":9, "bar":10}, 18, "baz"]';
var f1 = function(data) { return data[0].bar + 11;};
var f2 = JSON.parse;
f1(f2(jsdata)); // 10 + 11 === 21
var f1f2 = compose(f1, f2);
f1f2(jsdata); // 10 + 11 === 21


// 6. Dynamically defined scope in functions (this)
//
//    Now, what if we want to go one level more abstract? This time
//    we want to define a closure that doesn't take on its values
//    from the current context, but from some context that we will
//    dynamically specify at a later date.
var bb = 10;
var static_closure = function(aa) {
    return aa + bb;
};
static_closure(3);
// 13

var dynamic_closure = function(aa) {
    return aa + this.bb;
};

var context1 = {
    'fn': dynamic_closure,
    'bb': 10
};

context1.fn(3);
// 13

var context2 = {
    'fn': dynamic_closure,
    'bb': Math.random()
};

context2.fn(3);
// 3.656272369204089  [your value will be different]


//
// Great. So, to recap, the 'this' variable points to a single object or
// else is undefined. You can use it to leave the enclosing scope for a
// closure unspecified until the last moment when it's executed.
//
```

```
// If you can, you may want to simply pass in the object itself as an
// explicit variable, as shown below. Otherwise one can use 'this'.
var simpler_than_dynamic_closure = function(aa, obj) {
    return aa + obj.bb;
};

simpler_than_dynamic_closure(3, context2);
// 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:

```
// The concept of self-executing functions

// 1. Basic self-executing functions for encapsulation
//
// Suppose you have some JS code like this:
var foo = "inner";
console.log(foo);

// You worry that the external world may also have defined a foo variable.
// To protect your code, you can encapsulate it by putting it in a
// self-executing function.
var foo = "outer";
(function() {
```

```javascript
    var foo = "inner";
    console.log(foo);
})();
console.log(foo);

// This will print:
/*
inner
outer
*/

// Let's break this down bit by bit. The inside is just an anonymous
// function definition. Let's assign it to a variable.
var anon = function () {
    var foo = "inner";
    console.log(foo);
};

// We enclose this function definition in parentheses, and then immediately
// execute it by adding another two parens at the end. We also need a
// semicolon as the expression is ending.
(function() {
    var foo = "inner";
    console.log(foo);
})();

// 2. Passing in variables to a self-executing function
//
// Now that we've set up a wall, we can do something more sophisticated by
// passing in specific variables from the enclosing scope.
var bar = "allowed";
(function(aa) {
    var foo = "inner";
    console.log(foo + " " + aa);
})(bar);

// This prints
/*
inner allowed
*/

// 3. Passing in and receiving variables from a self-executing function.
//
// Now we'll return an object as well, converting the inside into a proper
// function.
var bar = "allowed";
var result = (function(aa) {
    var foo = "inner";
    var out = foo + " " + aa;
    console.log(out);
    return {"val": out};
})(bar);
console.log(result);

// This prints
/*
{ val: 'inner allowed' }
*/
```

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:

```javascript
#!/usr/bin/env node

var request = require('request'); // npm install request
var apiurl = 'http://nodejs.org/api/index.json';
var callbackfn = function(error, response, body) {
  if (!error && response.statusCode == 200) {
    console.log(body);
  }
};
request(apiurl, callbackfn);

// When executed, this produces the following
/*
{
  "source": "doc/api/index.markdown",
  "desc": [
    {
      "type": "list_start",
      "ordered": false
    },
    {
      "type": "list_item_start"
    },
    {
      "type": "text",
      "text": "[About these Docs](documentation.html)"
    },
    {
      "type": "list_item_end"
    },
    {
      "type": "list_item_start"
    },
    {
      "type": "text",
      "text": "[Synopsis](synopsis.html)"
    },
    {
      "type": "list_item_end"
    },
    {
      "type": "list_item_start"
    },
...
*/
```

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".

```
#!/usr/bin/env node

var request = require('request');
var apiurl = 'http://nodejs.org/api/index.json';
var fs = require('fs');
var outfile  = 'index2.json';
request(apiurl, function(error, response, body) {
    if (!error && response.statusCode == 200) {
        fs.writeFile(outfile, body, function (err) {
            if (err) throw err;
            var timestamp = new Date();
            console.log("Wrote %s %s", outfile, timestamp);
        });
    }
});
```

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.

```
#!/usr/bin/env node

var request = require('request');
var apiurl = 'http://nodejs.org/api/index.json';
var fs = require('fs');
var outfile  = 'index3.json';
setTimeout(function() {
    request(apiurl, function(error, response, body) {
        if (!error && response.statusCode == 200) {
            fs.writeFile(outfile, body, function (err) {
                if (err) throw err;
                var timestamp = new Date();
                console.log("Invoked first, happens second at %s", new Date());
                console.log("Wrote %s %s", outfile, timestamp);
            });
        }
    });
}, 3000);
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.

```
#!/usr/bin/env node

var request = require('request');
var apiurl = 'http://nodejs.org/api/index.json';
var fs = require('fs');
var outfile  = 'index4.json';

var cb_writefile = function (err) {
    if (err) throw err;
    var timestamp = new Date();
    console.log("Invoked first, happens second at %s", new Date());
    console.log("Wrote %s %s", outfile, timestamp);
};

var cb_parsebody = function(error, response, body) {
    if (!error && response.statusCode == 200) {
        fs.writeFile(outfile, body, cb_writefile);
    }
};

request(apiurl, cb_parsebody);
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.

```
#!/usr/bin/env node

var request = require('request');
var apiurl = 'http://nodejs.org/api/index.json';
var fs = require('fs');
var outfile  = 'index-scoped.json';

var make_request = function(apiurl, OUTFILE) {

    var cb_writefile = function (err) {
        if (err) throw err;
        var timestamp = new Date();
        console.log("Wrote %s %s", OUTFILE, timestamp);
    };

    var cb_parsebody = function(error, response, body) {
        if (!error && response.statusCode == 200) {
            fs.writeFile(OUTFILE, body, cb_writefile);
        }
    };

    request(apiurl, cb_parsebody);
};

make_request(apiurl, outfile);

// Because we appropriately scoped outfile,
// we have a much more general and easy to use function.
// We can now easily pass in another output file.
var another_outfile ='index-another.json';
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.

```
#!/usr/bin/env node

var uu = require('underscore');
var request = require('request');
var apiurl = 'http://nodejs.org/api/index.json';
var fs = require('fs');
var outfile  = 'index-parsed.json';

var data2name_files = function(data) {
    var module2obj = function(mod) {
        var modregex = /\[([^\]]+)\]\(([^\)]+)\)/;
        var match = modregex.exec(mod);
        return {'name': match[1], 'file': match[2]};
    };
    var notUndefined = function(xx) { return !uu.isUndefined(xx);};
    var modules = uu.filter(uu.pluck(data.desc, 'text'), notUndefined);
    return uu.map(modules, module2obj);
};

var body2json = function(body) {
    return JSON.stringify(data2name_files(JSON.parse(body)), null, 2);
};

var make_request = function(apiurl, OUTFILE) {

    var cb_writefile = function (err) {
        if (err) throw err;
        var timestamp = new Date();
        console.log("Wrote %s %s", OUTFILE, timestamp);
    };

    var cb_parsebody = function(error, response, body) {
        if (!error && response.statusCode == 200) {
            fs.writeFile(OUTFILE, body2json(body), cb_writefile);
        }
    };

    request(apiurl, cb_parsebody);
};

make_request(apiurl, outfile);

/* This code turns the raw output:

        {
          "source": "doc/api/index.markdown",
          "desc": [
            {
              "type": "list_start",
              "ordered": false
            },
            {
              "type": "list_item_start"
            },
            {
              "type": "text",
              "text": "[About these Docs](documentation.html)"

        ...

Into something like this:

        [
          {
            "name": "About these Docs",
            "file": "documentation.html"
          },
          {
            "name": "Synopsis",
            "file": "synopsis.html"
          },
        ...
]*/
```

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 similar 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 keyword[9] 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:

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

[9]You 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.

```
  host: 'example.com',
  port: '5432',
  dbpass: 'mydbpass' }
*/
```

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.

```
var dbobj = {
    protocol: 'postgres',
    user: 'myuser',
    pass: 'mypass',
    host: 'example.com',
    port: '5432',
    dbpass: 'mydbpass',
    toString: function() {
        return this.protocol + '://' +
            this.user + ':' +
            this.pass + '@' +
            this.host + ':' +
            this.port + '/' +
            this.dbpass;
    }
};

dbobj.toString();
// '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).

```javascript
// Example of inheritance
//
// Item - sku, price
// Book - title, text, search
// Furniture - name, width, length


// 0. Preliminaries: this helper function copies the properties and methods
// of the parentObject to the childObject using the copyOfParent
// intermediary.
//
// See Zakas' writeup at goo.gl/o1wRG0 for details.
function inheritPrototype(childObject, parentObject) {
   var copyOfParent = Object.create(parentObject.prototype);
   copyOfParent.constructor = childObject;
   childObject.prototype = copyOfParent;
}

// 1. Define the parent class constructor and add prototype methods
function Item(sku, price) {
    this.sku = sku;
    this.price = price;
}
Item.prototype.toString = function () {
    return "SKU: "+ this.sku + " | Price: " + this.price + " USD";
};

// 2. Define the subclass constructor, copy over properties and methods of
// Item, and then define a new function.
function Book(sku, price, title, text) {
    Item.call(this, sku, price);
    this.title = title;
    this.text = text;
};

inheritPrototype(Book, Item);

Book.prototype.search = function(regexstr) {
    var regex = RegExp(regexstr);
    var match = regex.exec(this.text);
    var out = '';
    if(match !== null) {
        var start = match.index;
        var end = match.index + match[0].length;
        var dx = 3;
        var padstart = start - dx > 0 ? start - dx : start;
        var padend = end + dx > 0 ? end + dx : end;
        out = '...' + this.text.slice(padstart, padend) + '...';
    }
    return out;
};

// 3. Do one more subclass for illustrative purposes
function Furniture(sku, price, name, width, length) {
    Item.call(this, sku, price);
    this.name = name;
    this.width = width;
    this.length = length;
};

inheritPrototype(Furniture, Item);

Furniture.prototype.floorArea = function() {
    return this.width * this.length;
};


//  4. Now let's test things out.
//  Remember to use new!
var foo = new Item("ID:8908308", 43.27);
foo.sku;                // 'ID:8908308'
foo.price;              // 43.27
foo.toString();         // 'SKU: ID:8908308 | Price: 43.27 USD'

var bible = new Book("ID:123456", 101.02, "The Bible", "In the beginning there was");
bible.sku;              // 'ID:123456'
bible.price;            // 101.02
bible.toString();       // 'SKU: ID:123456 | Price: 101.02 USD'
bible.search('there');  // '...ng there wa...'
bible.search('therex'); // ''

var chair = new Furniture("ID:79808", 2020.32, "A chair", .5, 4.2);
chair.sku;          // 'ID:79808'
chair.price;        // 2020.32
chair.toString();   // 'SKU: ID:79808 | Price: 2020.32 USD'
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:

```
// Inheritance isn't always the answer
//
// Instead of subclassing Honda, Mercedes, etc., we can often get away with
// switch statements or the like.  rather than going with a class as an
// extremely heavyweight if/then statement.
//
// NOTE: in this particular example, we can actually replace the switch
// statement with an object lookup, but in general one might have more
// sophisticated logic in each switch case.

function Car(make, model) {
    this.make = make;
    this.model = model;
}
Car.prototype.toString = function () {
    return "Make: "+ this.make + " | Model: " + this.model;
};
Car.prototype.serviceIntervals = function() {
    switch (this.make) {
    case 'Honda':
        out = [10000, 20000, 30000];
        break;
    case 'Mercedes':
        out = [20000, 40000, 60000];
        break;
    default:
        out = [5000, 10000, 20000];
        break;
    }
    return out;
};

// Now let's instantiate some variables.

var accord = new Car('Honda', 'Accord');
var merc = new Car('Mercedes', 'S-Class');
var misc = new Car('Chrysler', 'Eminem Ride');

```

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

```
// Inheritance isn't always the answer, pt. 2
//
// Let's do another example where we show two different kinds of objects
// relate via composition.
function Wheel(isfront, isright) {
    this.isfront = isfront;
    this.isright = isright;
}
Wheel.prototype.toString = function () {
    var pos1 = this.isfront ? 'f' : 'b'; // front/back
    var pos2 = this.isright ? 'r' : 'l'; // right/left
    return pos1 + pos2;
};
var wheel1 = new Wheel(true, true);
var wheel2 = new Wheel(true, false);
var wheel3 = new Wheel(false, true);
var wheel4 = new Wheel(false, false);

wheel1.toString(); // 'fr'
wheel2.toString(); // 'fl'
wheel3.toString(); // 'br'
wheel4.toString(); // 'bl'

// We use the invoke method in underscore.
// See underscorejs.org/#invoke
var uu = require('underscore');
function Car(make, model, wheels) {
    this.make = make;
    this.model = model;
    this.wheels = wheels;
}
Car.prototype.toString = function () {
    var jsdata = {'make': this.make,
                  'model': this.model,
                  'wheels': uu.invoke(this.wheels, 'toString')};
    var spacing = 2;
    return JSON.stringify(jsdata, null, spacing);
};

var civic = new Car('Honda', 'Civic', [wheel1, wheel2, wheel3, wheel4]);

/*
{ make: 'Honda',
  model: 'Civic',
  wheels:
   [ { isfront: true, isright: true },
     { isfront: true, isright: false },
     { isfront: false, isright: true },
     { isfront: false, isright: false } ] }
*/

console.log(civic.toString();)
/*
{
  "make": "Honda",
  "model": "Civic",
  "wheels": [
    "fr",
    "fl",
    "br",
    "bl"
  ]
}
*/
```

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 exists, 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.