How To Read Source Code

Aria Stewart

@aredridel

I almost didn't give this talk.

What do we mean when we say "reading source code"?

Reading for comprehension

... to find bugs

... to find interactions

... to review

... to see the interface

... to learn!

Reading isn't linear

Reading Order

Do you start at the entry point of a package?

How about in a browser?

Maybe find the biggest source code file and read that first.

Try setting a breakpoint early and tracing down through functions in a debugger.

Try setting a breakpoint deep in the code, and reading each function in the call stack.

Kinds of source code

- Javascript
- C++
- ES6
- Coffee Script
- Shell Script

- SNOBOL
- APL
- Befunge
- Forth
- Perl
- Other people's javascript

Another way to think of kinds of source code

- Glue
- Interface-defining
- Implementation
- Algorithmic
- Configuring
- Tasking

What's 'Glue'?

How do you read glue?

This is from Ben Drucker's `stream-to-promise`

```
internals.writable = function (stream) {
  return new Promise(function (resolve, reject) {
    stream.once('finish', resolve);
    stream.once('error', reject);
  });
};
```

More glue!

```
var record = {
    name: (input.name || '').trim(),
    age: isNaN(Number(input.age)) ? null : Number(input.age),
    email: validateEmail(input.email.trim())
}
```

What's interface-defining code?

From node's events.js

```
exports.usingDomains = false;

function EventEmitter() { }
exports.EventEmitter = EventEmitter;

EventEmitter.prototype.setMaxListeners = function setMaxListeners(n) { };
EventEmitter.prototype.emit = function emit(type) { };
EventEmitter.prototype.addListener = function addListener(type, listener) { };
EventEmitter.prototype.on = EventEmitter.prototype.addListener;
EventEmitter.prototype.once = function once(type, listener) { };
EventEmitter.prototype.removeListener = function removeListener(type, listener) { };
EventEmitter.prototype.removeAllListeners = function removeAllListeners(type) {};
EventEmitter.prototype.listeners = function listeners(type) { };
EventEmitter.listenerCount = function(emitter, type) { };
```

Implementation

```
startRouting: function() {
  this.router = this.router || this.constructor.map(K);

  var router = this.router;
  var location = get(this, 'location');
  var container = this.container;
  var self = this;
  var initialURL = get(this, 'initialURL');
  var initialTransition;

  // Allow the Location class to cancel the router setup while it refreshes
  // the page
  if (get(location, 'cancelRouterSetup')) {
    return;
  }

  this._setupRouter(router, location);

  container.register('view:default', _MetamorphView);
  container.register('view:toplevel', EmberView.extend());

  location.onUpdateURL(function(url) {
    self.handleURL(url);
  });

  if (typeof initialURL === "undefined") {
    initialURL = location.getURL();
  }
  initialTransition = this.handleURL(initialURL);
  if (initialTransition && initialTransition.error) {
    throw initialTransition.error;
  }
},
```

From Ember.Router

Process entailment

Or, looking forward: *How much is required to use this thing correctly?*

and backward: *If we're here, what got us to this point?*

Algorithms

```
function Grammar(rules) {
  // Processing The Grammar
  //
  // Here we begin defining a grammar given the raw rules, terminal
  // symbols, and symbolic references to rules
  //
  // The input is a list of rules.
  //
  // The input grammar is amended with a final rule, the 'accept' rule,
  // which if it spans the parse chart, means the entire grammar was
  // accepted. This is needed in the case of a nulling start symbol.
  rules.push(Rule('_accept', [Ref('start')]));
  rules.acceptRule = rules.length - 1;
```

This bit is from a parser engine I've been working on called `lotsawa`. More on the next slide.

```
// Build a list of all the symbols used in the grammar so they can be numbered instead of referred to
// by name, and therefore their presence can be represented by a single bit in a set.
function censusSymbols() {
  var out = [];
  rules.forEach(function(r) {
    if (!~out.indexOf(r.name)) out.push(r.name);

    r.symbols.forEach(function(s, i) {
      var symNo = out.indexOf(s.name);
      if (!~out.indexOf(s.name)) {
        symNo = out.length;
        out.push(s.name);
      }

      r.symbols[i] = symNo;
    });

    r.sym = out.indexOf(r.name);
  });

  return out;
}

rules.symbols = censusSymbols();
```

Reads like a math paper, doesn't it?

Configuration

```
app.configure('production', 'staging', function() {
  app.enable('emails');
});

app.configure('test', function() {
  app.disable('emails');
});
```

An example using Javascript for configuration.

```
"express": {
    "env": "", // NOTE: `env` is managed by the framework. This value will be overwritten.
    "x-powered-by": false,
    "views": "path:./views",
    "mountpath": "/"
},

"middleware": {

    "compress": {
        "enabled": false,
        "priority": 10,
        "module": "compression"
    },

    "favicon": {
        "enabled": false,
        "priority": 30,
        "module": {
            "name": "serve-favicon",
            "arguments": [ "resolve:kraken-js/public/favicon.ico" ]
        }
    },
```

A bit of kraken config file.

Configuration source code has some interesting and unique constraints that are worth looking for.

- Lifetime
- Machine writability
- Responsible people vary a lot more
- Have to fit in weird places like environment variables
- Often store security-sensitive information

Tasking

What do charging 50 credit cards, building a complex piece of software with a compiler and build tools, and sending 100 emails have in common?

Reading Messy Code

So how do you deal with this?

```
      DuplexCombination.prototype.on = function(ev, fn) {
    switch (ev) {
  case 'data':
  case 'end':
  case 'readable':
this.reader.on(ev, fn);
return this
  case 'drain':
  case 'finish':
this.writer.on(ev, fn);
return this
  default:
return Duplex.prototype.on.call(this, ev, fn);
    }
      };
```

You are seeing that right. That's reverse indendation. Blame Isaac.

Rose tinted glasses!

```
standard -F dc.js
```

```
DuplexCombination.prototype.on = function (ev, fn) {
  switch (ev) {
    case 'data':
    case 'end':
    case 'readable':
      this.reader.on(ev, fn)
      return this
    case 'drain':
    case 'finish':
      this.writer.on(ev, fn)
      return this
    default:
      return Duplex.prototype.on.call(this, ev, fn)
  }
}
```

It's okay to use tools while reading!

How do you read this?

```
(function(t,e){if(typeof define==="function"&&define.amd){define(["underscore","
jquery","exports"],function(i,r,s){t.Backbone=e(t,s,i,r)})}else if(typeof export
s!=="undefined"){var i=require("underscore");e(t,exports,i)}else{t.Backbone=e(t,
{},t._,t.jQuery||t.Zepto||t.ender||t.$)}})(this,function(t,e,i,r){var s=t.Backbo
ne;var n=[];var a=n.push;var o=n.slice;var h=n.splice;e.VERSION="1.1.2";e.$=r;e.
noConflict=function(){t.Backbone=s;return this};e.emulateHTTP=false;e.emulateJSO
N=false;var u=e.Events={on:function(t,e,i){if(!c(this,"on",t,[e,i])||!e)return t
his;this._events||(this._events={});var r=this._events[t]||(this._events[t]=[]);
r.push({callback:e,context:i,ctx:i||this});return this},once:function(t,e,r){if(
!c(this,"once",t,[e,r])||!e)return this;var s=this;var n=i.once(function(){s.off
(t,n);e.apply(this,arguments)});n._callback=e;return this.on(t,n,r)},off:functio
n(t,e,r){var s,n,a,o,h,u,l,f;if(!this._events||!c(this,"off",t,[e,r]))return thi
s;if(!t&&!e&&!r){this._events=void 0;return this}o=t?[t]:i.keys(this._events);fo
r(h=0,u=o.length;h<u;h++){t=o[h];if(a=this._events[t]){this._events[t]=s=[];if(e
||r){for(l=0,f=a.length;l<f;l++){n=a[l];if(e&&e!==n.callback&&e!==n.callback._ca
llback||r&&r!==n.context){s.push(n)}}}if(!s.length)delete this._events[t]}}retur
n this},trigger:function(t){if(!this._events)return this;var e=o.call(arguments,
1);if(!c(this,"trigger",t,e))return this;var i=this._events[t];var r=this._event
s.all;if(i)f(i,e);if(r)f(r,arguments);return this},stopListening:function(t,e,r)
{var s=this._listeningTo;if(!s)return this;var n=!e&&!r;if(!r&&typeof e==="objec
```

uglifyjs -b < backbone-min.js

```
(function(t, e) {
    if (typeof define === "function" && define.amd) {
        define([ "underscore", "jquery", "exports" ], function(i, r, s) {
            t.Backbone = e(t, s, i, r);
        });
    } else if (typeof exports !== "undefined") {
        var i = require("underscore");
        e(t, exports, i);
    } else {
        t.Backbone = e(t, {}, t._, t.jQuery || t.Zepto || t.ender || t.$);
    }
})(this, function(t, e, i, r) {
    var s = t.Backbone;
    var n = [];
    var a = n.push;
    var o = n.slice;
    var h = n.splice;
    e.VERSION = "1.1.2";
    e.$ = r;
    e.noConflict = function() {
```

Human Parts

Or: *guessing intent is dangerous but you learn so much*

There's a lot of tricks for figuring out what the author of something meant.

Look for guards and coercions

```
if (typeof arg != 'number') throw new TypeError("arg must be a number");
```

Looks like the domain of whatever function we're in is 'numbers'.

```
arg = Number(arg)
```

Coerce things to numeric. Same domain as above, but doesn't reject errors via exceptions. There might be `NaN`s though. Probably smart to read and check if there's comparisons that will be false against those.

Look for defaults

```
arg = arg || {}
```

Default to an empty object.

```
arg = (arg == null ? true : arg)
```

Default to true only if a value wasn't explicitly passed.

Look for layers

`req` and `res` from Express are tied to the web; how deep do they go?

Is there an interface boundary you can find?

Look for tracing

Are there inspection points?

Debug logs?

Do those form a complete narrative? Or are they ad-hoc leftovers from the last few bugs?

Look for reflexivity

Are identifiers being dynamically generated?

Is there `eval`? Metaprogramming? New function creation?

`func.toString()` is your friend!

Look at lifetimes

- Who initializes this?
- When does it change?
- Who changes it?
- Is this information somewhere else in the mean time?
- Is it ever inconsistent?

Look for hidden state machines

```
var isReadied = false;
var isFinished = false;
```

Or

```
START -> READY -> FINISHED
```

Look for hidden state machines

isReadied	isFinished	state
false	false	START
false	true	invalid
true	false	READY
true	true	FINISHED

Look for composition and inheritance

Is this made of parts I can recognize? Do those parts have names?

Look for common operations

`map`.

`reduce`.

cross-join.

It's time to go read some source code.

Enjoy!