

making your JavaScript debuggable

Patrick Mueller [@pmuellr](#), [muellerware.org](#)
senior node engineer at [NodeSource](#)

<http://pmuellr.github.io/slides/2015/11-debuggable-javascript>

<http://pmuellr.github.io/slides/2015/11-debuggable-javascript/slides.pdf>

<http://pmuellr.github.io/slides/> (all of Patrick's slides)

code reading

I'm doing 90% maintenance and 10% development, is this normal?

Stack Overflow

In 2001, more than 50% of the global software population is engaged in modifying existing applications rather than writing new applications.

Capers Jones

you will write a little code
you will read a lot of code
optimize for readability

pyramid of doom

```
fs.readdir(".", function(err, files){
  files.forEach(function(file) {
    fs.stat(file, function(err, stats){
      if (!stats.isFile()) return
      fs.readFile(file, "utf8", function(err, data){
        console.log(file, data.length)
      })
    })
  })
})
```

pyramid of doom fixed - I

```
fs.readdir(".", cbReadDir)
function cbReadDir(err, files) {
  files.forEach(eachFile)
}
function eachFile(file) {
  fs.stat(file, (err, stats) => cbStatFile(err, stats,
}
function cbStatFile(err, stats, file) {
  if (!stats.isFile()) return
  fs.readFile(file, "utf8", (err, data) => cbReadFile(e
}
function cbReadFile(err, data, file) {
  console.log(file, data.length)
}
```

pyramid of doom fixed - 2

```
fs.readdir(".", cbReadDir)
function cbReadDir(err, files) {
  files.forEach(eachFile)
}
function eachFile(file) {
  fs.stat(file, cbStatFile)
  function cbStatFile(err, stats) {
    if (!stats.isFile()) return
    fs.readFile(file, "utf8", cbReadFile)
  }
  function cbReadFile(err, data) {
    console.log(file, data.length)
  }
}
```

pyramid of doom - unnamed functions!

```
fs.readdir(".", function(err, files){  
  files.forEach(function(file) {  
    throw new Error("huh?")  
  })  
})
```

```
// Error: huh?  
//   at path/to/script.js:6:11  
//   at Array.forEach (native)  
//   at path/to/script.js:5:9  
//   at FSReqWrap.oncomplete (fs.js:82:15)
```


pyramid of doom - unnamed functions fixed!

```
fs.readdir(".", cbReadDir)
function cbReadDir(err, files) {
  files.forEach(eachFile)
}
function eachFile(file) {
  throw new Error("huh?")
}

// Error: huh?
//   at eachFile (path/to/script.js:9:9)
//   at Array.forEach (native)
//   at cbReadDir (path/to/script.js:6:9)
//   at FSReqWrap.oncomplete (fs.js:82:15)
```

pyramid of doom - see also

- [async - npm](#) - Caolan McMahon
- [Promises](#) - Axel Rauschmayer

linting and code style - **standard**

```
$ node_modules/.bin/standard
```

```
standard: Use JavaScript Standard Style
```

```
(https://github.com/feross/standard)
```

```
path/to/bole.js:1:22: Strings must use singlequote.
```

```
path/to/bole.js:3:18: Strings must use singlequote.
```

```
...
```

```
(it never ends)
```

No decisions to make. No .eslintrc, .jshintrc, or .jscsrc files to manage. It just works.

other things

- keep functions shorter than a "page"; v8 will "inline" short functions!
- one-line arrow functions - no return or braces needed!

```
[ 1, 4, 9 ].map(x => Math.sqrt(x)) // [ 1, 2, 3 ]
```

- lots of great general ideas in [Code Complete](#)

logging

The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.

Brian W. Kernighan

console.log()

```
console.log(__filename + ": foo")  
// prints: /path/to/script.js: foo
```

```
console.log("foo", "bar")  
// prints: foo bar
```

```
console.log({x:1, y:2})  
// prints: { x: 1, y: 2 }
```

```
console.log("a-%s-b %j", 1, {x:1})  
// prints: a-1-b {"x":1}
```

```
console.log(process)  
// prints: { title: 'node', ...many lines... }
```

console.time()

```
console.time("foo")
doStuff()
console.timeEnd("foo")

function doStuff() {
    // takes a long time
}

// prints: foo: 1121ms
```


console.trace()

```
function a() { b() }  
function b() { c() }  
function c() { console.trace("foo") }
```

a()

```
// Trace: foo  
//      at c (<program>:3:24)  
//      at b (<program>:2:16)  
//      at a (<program>:1:78)  
//      at ...
```

console.table()???

```
// dream code!
```

```
const people = [  
  {firstName: 'George', lastName: 'Bush'},  
  {firstName: 'Barack', lastName: 'Obama'},  
]
```

```
console.table(people)
```

```
// index   firstName   lastName  
// -----  
//      0   George    Bush  
//      1   Barack     Obama
```

logging that stays in your code

npm debug

```
const debugA = require("debug")("thing-A")
const debugB = require("debug")("thing-B")

function a() { debugA("thrashing") }
function b() { debugB("churning") }

setInterval(a, 500); setInterval(b, 333)
```

```
$ DEBUG=* node debug.js
thing-B churning +0ms
thing-A thrashing +0ms
thing-B churning +339ms
thing-A thrashing +501ms
...
```

npm winston

```
const winston = require("winston")

const transports = winston.transports

winston.remove(transports.Console)
winston.add(transports.Console, { level: "warn" })
winston.add(transports.File, { filename: "x.log" })

winston.info("info message")
winston.warn("warning message")
winston.error("error message")

// prints:
// warn: warning message
// error: error message
```

npm bunyan

```
const bunyan = require("bunyan")

const log = bunyan.createLogger({name: "myapp"})
log.level("info")

log.info("hi")

// prints
// {"name":"myapp", "hostname":"my-hostname",
//   "pid":49675, "level":30, "msg":"hi",
//   "time":"2015-10-27T03:49:14.759Z", "v":0}

// du -h bunyan - 2.5M
```

npm bole

```
const bole = require("bole")

const log = bole("myapp")
bole.output({ level: "info", stream: process.stdout })

log.info("hi")

// prints
// {"time":"2015-10-27T03:56:45.762Z",
//   "hostname":"my-hostname", "pid":53014,
//   "level":"info", "name":"myapp", "message":"hi"}

// du -h bole - 144K
```

npm hooker

```
function preCall(name) {  
  const args = [].slice.call(arguments, 1)  
  log("->", name, args)  
}  
function postCall(result, name) {  
  log("<-", name, result)  
}  
hooker.hook(Math, Object.getOwnPropertyNames(Math), {  
  passName: true,  
  pre: preCall,  
  post: postCall  
})  
  
Math.max(5, 6, 7)  
Math.sqrt(2)
```


npm hooker

prints:

```
-> Math.max: [5,6,7]  
<- Math.max: 7  
-> Math.sqrt: [2]  
<- Math.sqrt: 1.4142135623730951
```

also provides

- filtering arguments
- overriding results

error handling

builtin process events

```
process.on("exit", code =>
  console.log("exiting with code: " + code))
process.on("uncaughtException", err =>
  console.log("uncaught exception: " + err.stack))

function a() { throw new Error("die die die") }

a()

// prints:
//
// uncaught exception: Error: die die die
//   at a (.../script.js:9:22)
//   at Object.<anonymous> (.../script.js:11:1)
//   ... more stack trace lines
// exiting with code: 0
```

Error.prepareStackTrace() - before

```
try { a() } catch(err) { console.log(err.stack) }  
function a() { b() }  
function b() { c() }  
function c() { throw new Error("foo blatz") }  
  
// Error: foo blatz  
//       at c (.../script.js:5:22)  
//       at b (.../script.js:4:16)  
//       at a (.../script.js:3:16)  
//       at Object.<anonymous> (.../script.js:2:7)  
//       at Module._compile (module.js:456:26)  
//       ...
```

Error.prepareStackTrace() - after

```
Error.prepareStackTrace = function(err, stackTrace) {  
    ...  
}  
  
try { a() } catch(err) { console.log(err.stack) }  
function a() { b() }  
function b() { c() }  
function c() { throw new Error("foo blatz") }  
  
// Error: foo blatz  
//     script.js    13 - c()  
//     script.js    12 - b()  
//     script.js    11 - a()
```

Error.prepareStackTrace = ...

```
function v8PrepareStackTrace(error, callSites) {  
  for (let callSite of callSites) {  
    const funcName = callSite.getFunctionName()  
    const file      = callSite.getFileName()  
    const line      = callSite.getLineNumber()  
    ...  
  }  
  return outputString  
}
```

reference: [javascript_stack_trace_api.md](#)

npm longjohn - before

a()

```
function a() { setTimeout(b, 100) }  
function b() { setTimeout(c, 100) }  
function c() { throw new Error("foo") }
```

```
// Error: foo  
//       at c [as _onTimeout] (/path/to/script.js:6:22)  
//       at Timer.listOnTimeout [as ontimeout] (timers.js
```

npm longjohn - after

```
if (process.env.NODE_ENV !== 'production')
  require('longjohn')

a()
function a() { setTimeout(b, 100) }
function b() { setTimeout(c, 100) }
function c() { throw new Error("foo") }
// Error: foo
//       at [object Object].c (path/to/script.js:6:22)
//       at Timer.listOnTimeout (timers.js:92:15)
// -----
//       at [object Object].b (path/to/script.js:5:16)
//       at Timer.listOnTimeout (timers.js:92:15)
// -----
//       at a (path/to/script.js:4:16)
//       at Object.<anonymous> (path/to/script.js:2:1)
//       ...
```


actual debugging

builtin module repl

```
var repl = require("repl")

function a(i) {
  var context = repl.start("repl> ").context
  context.pi = 3.14
  context.arg = i
}

a(3)

// repl> pi
// 3.14
// repl> arg
// 3
// repl>
```

builtin debugger

```
function a() {  
  debugger  
  var x = 1  
  var y = 2  
  console.log(x + " + " + y + " = " + (x+y))  
}  
  
setTimeout(a, 1000)
```

builtin debugger

```
> node debug debugger.js
< debugger listening on port 5858
connecting... ok
...
debug> watch("x")
debug> cont
break in debugger.js:2
Watchers:
  0: x = undefined

1 function a() {
2     debugger
3     var x = 1
4     var y = 2
debug> next
```

```
...
debug> next
break in debugger.js:4
Watchers:
  0: x = 1

2     debugger
3     var x = 1
4     var y = 2
5     console.log(x + " + " ..
6 }
debug> cont
< 1 + 2 = 3
program terminated
debug>
```

npm node-inspector

- Chrome Dev Tools user interface
 - breakpoints
 - stepping
 - watches
- but for debugging node

IDEs with debugging support

- IntelliJ IDEA
- Visual Studio Code

cpu profiles / heap snapshots

- V8 provides a built-in sampling cpu profiler
 - see time spent in expensive functions
 - shows stack for those functions
- V8 provides a built-in heap snapshot facility
 - dumps a representation of **ALL** JS objects

cpu profiles / heap snapshots

- npm **v8-profiler**
- StrongLoop
- N|Solid

These tools generate files that can be loaded in Chrome Dev Tools. StrongLoop and N|Solid also provide their own viewers.

cpu profiles - pro tips

- **NAME YOUR FUNCTIONS**
- use **node --no-use-inlining** if your functions are getting inlined
- more info at [Google's Chrome DevTools site](#)

heap snapshots - pro tips

- data is organized by class name
- if classes won't work, inject "tags" (named class instances) into objects you want to track
- take two snapshots, then "Comparison view" to see object counts diffs between the two
- more info at [Google's Chrome DevTools site](#)

heap snapshots - tagging things

```
class RequestTag {}  
class ResponseTag {}  
...  
function requestHandler(req, res) {  
  req.__hstag = new RequestTag  
  res.__hstag = new ResponseTag  
  ...  
}
```

Now you can search the snapshot for "**tag**" to see all tagged objects.

demo - memory leak

- server that leaks **request** objects - [demos/snapshot-untagged.js](#)
- same server, but tags **request** and **response** objects - [demos/snapshot-tagged.js](#)
- run both, take heap snapshots, and you can see from the 'tagged' version exactly what's leaking, since **requests** are instances of **IncomingMessage**

demo - cpu profiling

- program with a number of small functions - [demos/profile-inline.js](#)
- run with no special flags - most of the functions will be inlined, and no longer visible in stack traces
- run with **--no-use-inlining** - none of the functions will be inlined, and all will be visible in stack traces

how can you help?

write debugging tools!

- lots of low hanging fruit
 - what do other languages support?
 - what did Smalltalk and LISP do 30 years ago?
- lots of data from existing v8 debugging tools
- also needed - better typesetting of code

cpu profiles

- tree of function invocation records
- for each function
 - functions called, available as **children**
 - number of "ticks" spent executing code

Highlight expensive functions/lines in your editor?

Find which module is uses the most CPU?

heap snapshots

- **tons** of data; even a small snapshot is megabytes in size
- start with **npm snapshot-utils**
- example: displaying all "variables" in a snapshot

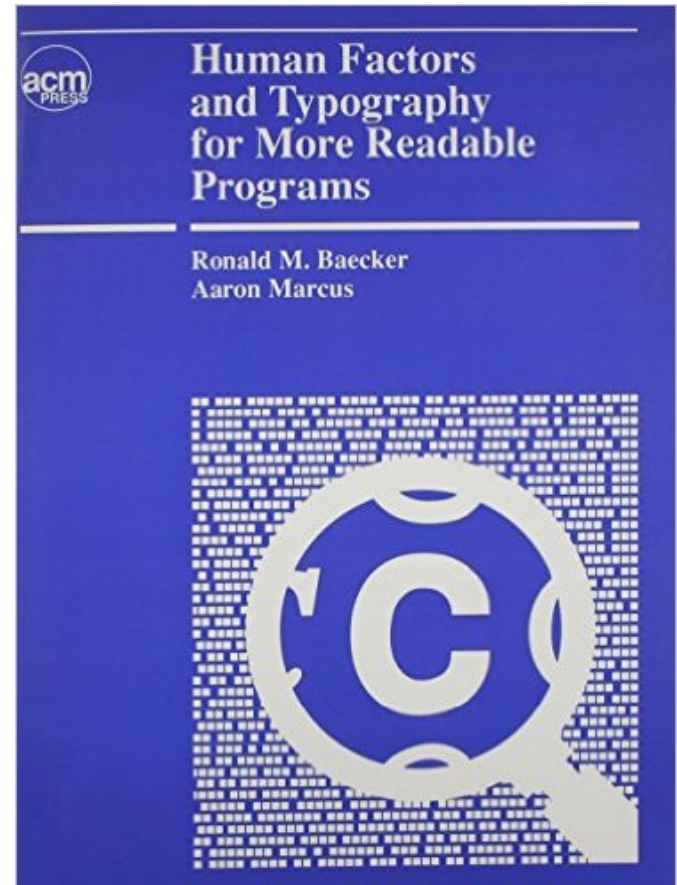
how can you help?

making your JavaScript debuggable

Human Factors and Typography for More Readable Programs

- 1990
- Ronald M. Baecker, Aaron Marcus

ISBN 0201107457



Encode phone number as a vector of digits, without punctuation. Returns number of digits in phone number or FALSE to indicate failure.

static bool

getpn(str)

char	*str;
int	i = 0;

```
while (*str != '\0')
    if (i >= PNMAX)
        return FALSE;
```

Set pn to the digits ignoring spaces and dashes

```
if (*str != ' ' && *str != '-')
    if ('0' <= *str && *str <= '9')
        pn[i++] = *str - '0';
    else
        return FALSE;
```

Fig. 1. A program presentation example from Baecker/Marcus [1, pg. 61]

fin