Minutes of the:	***42nd meeting of Ecma TC39***
in:	***Boston, MA, USA***
on:	***23-25 September 2014***

1 Opening, welcome and roll call

1.1 Opening of the meeting (Mr. Neumann)

Mr. Neumann has welcomed the delegates at Nine Zero Hotel (hosted by Bocoup) in Boston, MA, USA.

Companies / organizations in attendance:

Mozilla, Google, Microsoft, Intel, eBay, jQuery, Yahoo!, IBM; Facebook, IETF

1.2 Introduction of attendees

John Neumann – Ecma International

Erik Arvidsson – Google

Mark Miller – Google

Andreas Rossberg - Google

Domenic Denicola – Google

Erik Toth – Paypal / eBay

Allen Wirfs-Brock – Mozilla

Nicholas Matsakis – Mozilla

Eric Ferraiuolo – Yahoo!

Caridy Patino – Yahoo!

Rick Waldron – jQuery

Yehuda Katz – jQuery

Dave Herman – Mozilla

Brendan Eich (invited expert)

Jeff Morrison – Facebook

Sebastian Markbage – Facebook

Brian Terlson – Microsoft

Jonathan Turner – Microsoft

Peter Jensen – Intel

Simon Kaegi – IBM

Boris Zbarsky - Mozilla

Matt Miller – IETF (liaison)

1.3 Host facilities, local logistics

On behalf of Bocoup **Rick Waldron** welcomed the delegates and explained the logistics.

1.4 List of Ecma documents considered

2014/038	Minutes of the 41st meeting of TC39, Redmond, July 2014
2014/039	TC39 RF TG form signed by the Imperial College of science technology and medicine
2014/040	Venue for the 42nd meeting of TC39, Boston, September 2014 (Rev. 1)
2014/041	Agenda for the 42nd meeting of TC39, Boston, September 2014 (Rev. 1)
2014/042	Draft Standard ECMA-262 6th edition, Rev. 27, August 24
2014/043	TC39 RF TG form signed by Indiana University
2014/044	Responses to the Ecma Contribution License Agreement (CLA), 17 September 2014
2014/045	Presentation "Problem: exposing uninitialized built-in objects"
2014/046	Presentation "Object Instantiation Redo"
2014/047	Presentation "ES6 Rev27"
2014/048	Presentation "Trailing Commas"
2014/049	Presentation "Types and Type Annotations"

2 Adoption of the agenda (2014/041-Rev1)

Final Agenda for the: 42st meeting of Ecma TC39

```
in: Boston, Massachusetts, USA
on: 23 - 25 Sept. 2014
TIME: 10:00 till 17:00 EDT on 23rd and 24th of Sept. 2014
      10:00 till 16:00 EDT on 25th of Sept. 2014
LOCATION:
    Nine Zero Hotel
    90 Tremont St,
    Boston, MA 02108
    617-772-5800
```

1. ✓ Opening, welcome and roll call
   i. Opening of the meeting (Mr. Neumann)
   ii. Introduction of attendees
   iii. Host facilities, local logistics
2. ✓ Adoption of the agenda (2014/041)
3. ✓ Approval of the minutes from July 2014 (2014/038)
4. ECMA-262 6th Edition
   i. ✓ Spec. status report (Allen)

ii. ✓ Modules status report

iii. ✓ Instantiation Reform (Allen). Review of new design. Consider some detail alternatives. See (https://gist.github.com/allenwb/5160d109e33db8253b62) and (https://gist.github.com/allenwb/53927e46b31564168a1d)

iv. ✓ Legacy *decimal* integer literals starting with 0 and containing 8 or 9. (Jason Orendorff, Boris Zbarsky)

v. ✓ `Number('0b0101').NaN` or not? https://bugs.ecmascript.org/show_bug.cgi?id=1584 (Erik Arvidsson)

vi. ✓ More Function-in-Block (Brian)

5. ECMA-262 7th Edition

   i. ✓ Trailing commas in CallExpressions (Jeff Morrison)

   ii. ✓ Flush demorm to zero (Allen)

      a. http://esdiscuss.org/topic/float-denormal-issue-in-javascript-processor-node-in-web-audio-api

      b. https://bugzilla.mozilla.org/show_bug.cgi?id=1027624#c30

   iii. ✓ ArrayBuffer.transfer (Brendan/Allen for Luke Wagner)

      a. https://gist.github.com/andhow/95fb9e49996615764eff

   iv. ✓ Types (Jonathan, Brian)

   v. ✓ Math.iaddh, etc. polyfillable 64-bit int helpers (Brendan for Fabrice Bellard)

      a. http://esdiscuss.org/topic/efficient-64-bit-arithmetic

      b. https://gist.github.com/BrendanEich/4294d5c212a6d2254703

   vi. ✓ NoSuchProperty built-in proxy (Brendan for Nicholas C. Zakas)

      a. http://esdiscuss.org/topic/my-ecmascript-7-wishlist

   vii. ✓ Object Rest Destructuring and Spread Properties (Sebastian)

      a. https://github.com/sebmarkbage/ecmascript-rest-spread

   viii. ✓ Exponentiation Operator Update (Rick)

      a. Stage 1: https://github.com/rwaldron/exponentiation-operator/issues/1

      b. Stage 2: https://github.com/rwaldron/exponentiation-operator/issues/2

   ix. ✓ RegExps that don't modify global state (Domenic and Jaswanth)

   x. ✓ global.asap for enqueuing a microtask (Domenic and Brian)

   xi. Array.prototype.contains advance to stage 2 (Domenic)

6. ✓ Test 262 Status

7. ECMA-402 2nd Edition

   i. Status Update (Rick)

a. This item is pending an offline progress review.

8. ✓ Report from the Ecma Secretariat

9. Date and place of the next meeting(s)

   i. November 18 - 20, 2014 (San Jose - PayPal)

   ii. January 27 - 29, 2015 (San Francisco - Mozilla)

   iii. March 24-26, 2015 (Europe)

   iv. May 27 - 29, 2015 (San Francisco - Yahoo)

   v. July, 28 - 30, 2015 (Redmond, WA - Microsoft)

   vi. September 22 - 24, 2015 (Portland, OR - jQuery)

   vii. November 17 - 19, 2015 (San Jose - PayPal)

10. Group Work Sessions

    i. Make suggestions for post meeting work sessions to be held on last day of meeting

    ii. (None)

11. Closure

The agenda was approved.

3 Approval of the minutes from the July 2014 Meeting (2014/038)

2014/038, the minutes of the 41th meeting of TC39, Redmond, July 2014 were approved without modification.

4 On the work of the TC39 RF (Royalty Free) Task Force – "Opt-out" of ECMA-262 (2014/031, 2014/034)

The 1st Opt-out window was still open at the time of the Boston TC39 meeting (between Aug. 11- October 11, 2014). During this time RFTG members could notify the Ecma Secretariat if they want to out for certain parts of the 2014/031 (ECMA-262 draft of July 2014) from RF to RAND or no-license mode.

5 For details of the technical discussions see Annex 1

6 Status Reports

6.1 Report from Geneva

Nothing new at this meeting.

6.2 Json

Matt Miller from Cisco has been approved as liaison manager between IETF and Ecma TC39. He took part in the TC39 Boston meeting.

7 Date and place of the next meeting(s)

Schedule 2014 meetings:

- November 18 - 20, 2014 (San Jose, CA - PayPal)

Schedule 2015 meetings:

- January 27-29, 2015 (San Francisco, Mozilla)
- March 24-26, 2015 (Paris, France; Google)
- May 27-29, 2015 (Menlo Park, Facebook)
- July 28-30, 2015 (Redmond, Microsoft)
- Sept. 22-24, 2015 (Boston, Bocoup)
- Nov. 17-19, 2015 (Paypal, San Jose)

8 Any other business

None.

9 Closure

Mr. Neumann thanked **Bocoup** for hosting the meeting in Boston, the TC39 participants their hard work, and **Ecma International** for holding the social event dinner Wednesday evening. Special thanks goes to **Rick Waldron** to take the technical notes of the meeting.

Annex 1

From: Rick Waldron <waldron.rick@gmail.com>

Subject: Sept 2014 Meeting Notes and Slides

# September 23 2014 Meeting Notes

Brian Terlson (BT), Allen Wirfs-Brock (AWB), John Neumann (JN), Rick Waldron (RW), Eric Ferraiuolo (EF), Jeff Morrison (JM), Jonathan Turner (JT), Sebastian Markbage (SM), Istvan Sebestyen (phone) (IS), Erik Arvidsson (EA), Brendan Eich (BE), Domenic Denicola (DD), Peter Jensen (PJ), Eric Toth (ET), Yehuda Katz (YK), Dave Herman (DH), Brendan Eich (BE), Simon Kaegi (SK), Boris Zbarsky (BZ), Andreas Rossberg (ARB), Caridy Patino (CP), Niko Matsakis (NM), Mark Miller (MM), Matt Miller (MMR), Jaswanth Sreeram (JS)

# Introduction

RW: (explanation of evening meetup)

TC39 expresses its appreciation to Bocoup for hosting and Ecma for dinner.

JN: Adoption of agenda?

Approved: https://github.com/tc39/agendas/blob/master/2014/09.md

JN: Approval of minutes (July 2014)

#### Conclusion/Resolution

- Agenda approved
- Previous Minutes approved

## 4.1 Spec status report

(Allen Wirfs-Brock)

Rev27meeting_review.pdf

AWB:

- Added %IteratorPrototype% that all built-in iterators inherit from.

Issue:

Should %IteratorPrototype% define an @@iterator method?

- Definition: return this value
- Currently defined in %GeneratorPrototype% and in the prototype for all built-in iterators.
- https://bugs.ecmascript.org/show_bug.cgi?id=3104

DD: This supports the established convention

YK: There was objection from Jafar
DH: but that was to iterator as self-iterable in whole, not this Iterator.prototype which we want

AWB: (summary in slides)

- Destructuring performs ToObject
- Array and generator comprehensions removed
- Realm removed
- Revised @@unscopable
- Object.assign ignores null and undefined sources
- An error to reentrantly enter generator via next/throw/return
- String(sym) now returns the symbol's description. Does not change ToString(symbol) throwing
- `sym === Object(sym)` is now true
- Tighten specification for when `Array.prototype.sort` is undefined

AWB: Things being removed should be added to https://github.com/tc39/ecma262

EA: @@unscopables shipping in v8 (needed for Array.prototype.values)

AWB:

- Added 16.1 that lists forbidden extensions

- [Discussion](https://github.com/rwaldron/tc39-notes/blob/master/es6/2014-07/jul-29.md#49-argumentscaller-poisoning-on-new-syntactic-forms---arrows-generators)

- Removed all explicit poison pill methods on functions (except Function.prototype)

[16.1 Forbidden Extensions](https://people.mozilla.org/~jorendorff/es6-draft.html#sec-forbidden-extensions)

AWB: Mark's concern was stack walking capabilities

DH: At odds with proper tail calls

AWB: In ES5 we added things to spec PTC

YK: Of note: the new JDK's JavaScript engine is claiming to be "spec compliant", but implements racy threads.

DH: Most of the algorithms will fall down because they're not designed for threaded environments

Should ECMA-262 section 16 say something about cases like this?

AWB: We can express the "intent", that these are meant to be run atomically

DH: Memory models are hard.

ARB: Just say something informally to the effect that all algorithms in this spec are assumed to be executed atomically, i.e., are never observably interleaved with, or concurrent with, any other mutation of the programs state.

ARB: What specifically does "Extensions" refer to?

AWB: Specifically the definition in [16](https://people.mozilla.org/~jorendorff/es6-draft.html#sec-error-handling-and-language-extensions), bullet 2.

AWB: w/r to Mark's concerns, we may want to specifically say that we don't want implementors to implement stack walking outside of devtools.

AWB: Open work items:

- Update Introduction (BE)
- Update Language Overview (RW)
- RegExp normative and Annex B issues
- Lexical Grammar may need work
- Annex B spec for HTML style `<!-- -->` comments
- Module spec work closing in
- Instantiation reform, language + built-ins

AWB: Is anyone interesting in analysing the changes in ES5 to determine things that may have been wrong and need to be rolled back?

RW: Should reach out to Matthias Bynens

AWB: Ideally someone willing to become fully aware of the issues and propose/execute on solutions (if necessary)

AWB: Outline of final spec review process from January through June.

JN: Confirm RFTG opt-out

RW: https://github.com/rwaldron/tc39-notes/blob/master/es6/2014-07/jul-30.md#rftg-admin-es6-opt-out-period

#### Conclusion/Resolution

- Add `Symbol.iterator` on the %IteratorPrototype% that returns `this`.
- Due for November meeting
- Language Overview draft (RW)
- Introduction (BE)

## 4.3 Legacy Decimal Integer Literals starting with 0 and containing 8 or 9.

(Allen Wirfs-Brock, Jason Orendorff, Boris Zbarsky)

http://esdiscuss.org/topic/early-error-on-0-followed-by-8-or-9-in-numeric-literals-does-not-seem-to-be-web-compatible

AWB: Octals beginning with `0` are restricted in strict mode

BZ: Credit card and telephone numbers

AWB: Dates

AWB: In non-strict, where `0` normally means an octal, unless it contains an 8 or 9.

MM: Any program that relies on this will break

```js
> 055
45
> 089
89
> 051082
51082
> (051082).toString()
"51082"
> (051078).toString()
"51078"
> (051077).toString()
"21055"
```

(Discussion about use of broken features)

MM: Want to preserve the restriction as currently written in strict mode. Strict mode program, testing during August, evals `08` will pass, in January it will not.

AWB: Any browser incompatibilities?

BZ: Anyone writing strict mode code won't be writing code like this. For strict mode there is no hazard.

AWB: For Annex B?

BZ: Pull

MM: All strict mode behaviour is normative

AWB: + Annex B

Discussion re: _EscapeCharacter_

https://bugs.ecmascript.org/show_bug.cgi?id=3212

BE: It was changed in ES3 to use DecimalDigit (from ES1's OctalDigit), would need to ask Waldemar

#### Conclusion/Resolution

- Leave strict mode as-is
- Disallow leading zeroes, except for the constant 0 (and 0.1234 etc)
- Annex B non-strict: explicitly define the current interoperable behavior
- `\8` and `\9` are equivalent to literal 8 and 9
- '\08' is NUL char and then '8', '\08'.length is 2

## 4.4 Number('0b0101'). NaN or not?
(Erik Arvidsson)

EA: Previous discussion: https://github.com/rwaldron/tc39-notes/blob/c61f48cea5f2339a1ec65ca89827c8cff170779b/es6/2014-04/apr-9.md#46-updates-to-parseint

Should `Number` be able to parse the string "0b0" or "0o1"

(Discussion of people (ab)using Number for converting user input and whether this should affect things.)

Yes.

#### Conclusion/Resolution

- Use spec-internal `ToNumber` via userland `Number` called as a function will convert (ie `Number('0b101') === 5)`.
- Upholding previous consensus on `parseInt`
- Note in Annex C that this extension is a potentially breaking change

## 4.5 More Function-in-Block
(Brian Terlson)

BT: (covers current spec)

AWB: Exactly the strict modes semantics of functions in blocks

BT: Don't hoist past/up-to nearest function

New issue:

```js
if (cond) {
  function F() {}
  function G() { F() }
}

// sometime later
if (cond) {
  function F() {}
}

G();
```

In today's semantics, when `G();` is called, it calls the second `F();`. In the new design, it would be the first, which is a breaking change. This code actually exists in some ad code somewhere.

BT: No one else has implemented this yet, need to know if IE will be out there alone, or will other browsers stand with IE and break that unusual ad code?

DH: If v8 and spidermonkey post bugs that indicate the intention to implement, then it shows support among implementors and is valuable.

#### Conclusion/Resolution

- No spec change
- Implementors will file intention commitment bugs
  [V8 bug](https://code.google.com/p/v8/issues/detail?id=3594)
  [SpiderMonkey bug](https://bugzilla.mozilla.org/show_bug.cgi?id=1071646)

## Intro to Matt Miller from IETF (Liaison manager)

MMR: (Role as liaison from IETF, to establish a productive relationship, specifically w/r to JSON)

AWB: IETF Documents, e.g. JavaScript mimetype references to ES3. Far behind considering ES5 and soon ES6

MMR: The RFCs are stable documents. When a document receives a number, it's set. If it needs to be updated, it can be replaced.

Discussion about interoperability and incompatibility breaks.

#### Conclusion/Resolution

- Improved communication

## Revisiting 16.1, Forbidden Extensions

MM: SpiderMonkey removed own caller and own arguments from all function forms
... Needs to be clear: if the this binding is strict or built-in.

AWB: interpret: If it isn't a non-strict function, it behaves exactly as spec'ed.

Discussion of strict/non-strict w/r to built-ins

MM: Applied to anything other than a non-strict function?

(10 minute back and forth about the definitions of "strict function" and "ECMAScript function")

MM's overall concern is to make sure the behavior is nailed down for built-ins as well as for strict functions.

## Somehow we started talking about test262

Discussion of test262's (lack of) coverage.

BT: We didn't have coverage in test262 of a non-enumerable property shadowing an enumerable property.

JM: it would be good to have a place to list the open coverage holes.

SK/DD: while reviewing sections of the spec, people should write down a list of test cases that should be written (e.g. https://github.com/domenic/Array.prototype.contains/issues/1)

AWB: Perhaps we should try to get a rough statement of what our coverage areas are. Some areas are covered because they were in ES3 tests; Microsoft contributed a lot of standard library tests; there are probably lots of gaps.

(Discussion of running test262 tests against implementations, especially given their optimizations.)

YK: ideally we should not run the tests in interpreted mode.

AR: there is no hot code in test262, so right now we only run in interpreted mode.

YK: could we use switches to turn up optimizations?

AR: there are complications that in V8 at least make that unlikely to be useful.

(Back to topic of a statement of coverage)

AWB: I'm mostly interested in whether there is a useful set of tests for most areas of the spec

BT: was this the test262 agenda item?

BE: this was the "depress the room" agenda item

DD/BT: (discussion of poisoning built-ins before running tests)

YK: Value of poisoning?

BT: Implementations have to take care to not rely on user modifiable built-ins.

- Test262 attempts to walk a balance, if there are known examples that break runtime, there will be tests, but we don't go looking for them.

#### Conclusion/Resolution

- It's impossible to test ECMAScript
- Testing is hard

## Test 262 Update

(Brian Terlson)

BT: A number of PRs remain open from the Test The Web Forward event: people haven't signed CLA despite request, haven't updated patch according to review feedback. These will be closed.

- Browser will currently hang on 2**32 ToLength tests; these are initializing and iterating
- Previous consensus was to not update test262 on ecmascript.org. Any objections to creating a preview of ES6 test262?
- extracted test runner harness, wip.
- Suggest transferring test harness to Ecma

DD: Have two official runners?

DD/BT: Deprecating the python runner?

ARB: Have to check with testing infrastructure

DD: But can we "deprecate", in the

- Repository organization?

BT: Move all tests from present locations to new homes in new directory hierarchy
- someone, or something, needs to move 11,500 tests from their current location to their new location
- Leave all ES5 where they are?
- Create new tests under named feature.

AWB: As tests are updated, they could be moved?

BT: confirm

BT: who moved the ES3 tests to ES5 organization?

MM: that was me

BT: that was fun?

MM: I have experienced traumas in my life that were less fun

(Discussion of whether we should move the ES5 stuff piecemeal, e.g. as they get fixed.)

BT: It might be OK to move by feature, e.g. all the Array.prototype.forEach tests could move at once.

(Discussion of how to incorporate post-ES6 tests into the repo, as they are written, but before ES6 tests are finalized.)

AWB: when running "the ES6 test suite," it should only be things that are officially in ES6. But the JS community cares about what they can use in a browser. We should not create impressions that things are done or stable until they have worked through the process.

AR: so we should put experimental things into a separate directory

YK: no. You don't do that in Chrome; you use feature flags.

(Discussion of folder vs. feature flags for new test262 tests for post-ES6 experimental features.)

AWB: ok with stage 3 being in the repo

DD: needs to be stage 1 and onward

AWB: At stage 1, it's just the champion

BT: Prerequisite to stage 3: have a PR open with your test

DD: Just need a way to flag something as experimental

BT: Accept Allen's proposal?

DD/YK: nooooo

ARB: Fine to have early stage features, as long as they're not intermingled.

BT: The file system isn't ideal.

(Discussion of whether to accept stage 1 onward into test262.)

DD: Want to make test262 writing easy, so will be adding helpers.

#### Conclusion/Resolution

- Create a subdomain to host test262's ES6 tests
- Deprecation plan for Python harness: contact
- Domenic or Andreas for V8
- Dave for SpiderMonkey
- Overturn previous consensus, leave ES5 tests where they are.
- Let Brian decide which features to pull in
- Stage 3 features (and onward) shows up on the experimental public site
- Add helpers, that's good.

## 5.1 Trailing Commas in Function Call Expressions and Declarations

(Jeff Morrison)

JM: Useful for diffs in version control systems, especially Git blame. In other languages they often do a comma at the end of each line. We do for commas too.

BE: arrays aren't parameter lists. What other languages allow this?

JM: Python, D, Hack, probably others

JM: you don't have to do this if you don't want to; it's easy to lint for...

MM: just use comma-first style?

JM: doesn't work well for the first argument

AWB: What about rest parameter position? Should it be a syntax error after a rest param?

JM: Hadn't thought about it. No strong feelings. Makes sense to me to be invalid after a trailing rest param

```js
function f(
a,
b,
...c
   ) {
return 1;
}
```

MM: what's the argument against this?

JM: if people use this, old engines won't be able to run that code.

AWB: it's the "no new syntax" argument.

BE: the funny thing is in C you can't have enums with trailing commas but you can have them in array and struct initializers. That was a botch.

#### Conclusion/Resolution

BE: let's sleep on it

- No decision yet, revisit tomorrow.

## ?.? Process

AWB: When something reaches stage 1 we should create a repo at TC39. And use that issue tracker for discussion instead of es-discuss or bugs.ecmascript.org or other ad-hoc locations.

## 5.2 Flush denorm to zero
(Allen Wirfs-Brock)

http://esdiscuss.org/topic/float-denormal-issue-in-javascript-processor-node-in-web-audio-api
https://bugzilla.mozilla.org/show_bug.cgi?id=1027624#c30

AWB/BE: Explanation of "flush to denorm"

https://en.wikipedia.org/wiki/Denormal_number

Denorm values appear in audio algorithms. Modes exist to prevent producing denorms.

MM: Not just wasted precision:

```
|-|-|-|-------|-------|-|-|-|
    ^      0      ^
largest              smallest
negative             positive
normal               normal
```

AWB: Specifically desirable from people writing audio algorithms in JavaScript.

BE:

```js
Math.denormz(x);
Math.fdenormz(x);
```

```js
Math.fdenormz(x) === Math.denormz(Math.fround(x));
```

PJ: Another suggestion to have an FPU mode-setting function?

BE/AWB: That's the big red switch.

BE: Leads to all black-box/other-module functions needing save/restore advice around calls from your module

- Stage 1?

#### Conclusion/Resolution

- Stage 1

## 5.3 ArrayBuffer.transfer
(Brendan Eich, Allen Wirfs-Brock for Luke Wagner)

https://gist.github.com/andhow/95fb9e49996615764eff

AWB: (summarizes rationale)

(Discussion re detaching, immutability, freezing, and how they may or may not be related)

BE: ArrayBuffer.transfer seems safe, considering.

Discussion of whether it should be static on ArrayBuffer or a method on the prototype - it's an alternative constructor

Suggestion that ArrayBuffer.transfer(b1, 0) return a zero-length b2, not null. Consensus favors.

AWB: Subclass of `ArrayBuffer`? What is the class of the object returned by `transfer(...)`?

#### Conclusion/Resolution

- Stage 1

## 5.6 NoSuchProperty built-in proxy
(Brendan Eich for Nicholas C. Zakas)

BE: Want to be able to throw if a property access occurs for something that doesn't exist.

```js
var NoSuchProperty = Proxy({}, {
  has(target, name) {
   return true;
  },
  get(target, name, receiver) {
    if (name in Object.prototype) {
      return Object.prototype[name];
    }
    throw new TypeError(name + " is not a defined property");
  }
});
```

BE: JS1 was a rush, didn't have time for try/catch, so I was very shy about errors. This should be something that can be made an error. Too much fail-soft where things should've error'ed.

MM: If it's doable in library code, then we should wait until we see more activity.

YK/DD/RW: There are more Proxy related patterns that would more valuable to address before NoSuchProperty, considering it's possible in library code.

DH: (comments on likelihood of actual use)

BE: This will be slow

BZ: It will be fine until you hit the Proxy, based on real engine DOM (nodelist etc.) experience.

#### Conclusion/Resolution

- Keep in library code which others should develop and user-test at scale

## 5.8 Exponentiation Operator Update

(Rick Waldron)

RW: Presenting updates to feature proposal:

- Stage 1: https://github.com/rwaldron/exponentiation-operator/issues/1
- Stage 2: https://github.com/rwaldron/exponentiation-operator/issues/2

Progress to Stage 2?

Confirm.

#### Conclusion/Resolution

- Stage 2

## 5.5 Math.iaddh, etc. polyfillable 64-bit int helpers

(Brendan Eich for Fabrice Bellard)

http://esdiscuss.org/topic/efficient-64-bit-arithmetic
https://gist.github.com/BrendanEich/4294d5c212a6d2254703

MM: Recognizes "bigit" as good for bignum big decimal etc. digit, not just for int64 emu

(http://hackage.haskell.org/package/double-conversion-0.2.0.6/src/double-conversion/src/bignum.cc, http://code.ohloh.net/file?fid=J9lbBTlP__EhaF2Zrwq6Sk38lZE&cid=891oM4GeZTE&s=&fp=282835&mp&projSelected=true#L0)

Concrete proposal:

```js
Math.imulh();
Math.imuluh();
Math.iaddh();
Math.isubh();
```

#### Conclusion/Resolution

- Rename imuluh to umulh
- Stage 0

# September 24 2014 Meeting Notes

Brian Terlson (BT), Allen Wirfs-Brock (AWB), John Neumann (JN), Rick Waldron (RW), Eric Ferraiuolo (EF), Jeff Morrison (JM), Jonathan Turner (JT), Sebastian Markbage (SM), Istvan Sebestyen (phone) (IS), Erik Arvidsson (EA), Brendan Eich (BE), Domenic Denicola (DD), Peter Jensen (PJ), Eric Toth (ET), Yehuda Katz (YK), Dave Herman (DH), Brendan Eich (BE), Simon Kaegi (SK), Boris Zbarsky (BZ), Andreas Rossberg (ARB), Caridy Patino (CP), Niko Matsakis (NM), Mark Miller (MM), Matt Miller (MMR), Jaswanth Sreeram (JS)

## Object Instantiation Redo

(Allen Wirfs-Brock)

instantiation-reform-sept2014.pdf

AWB: (introducing discussion from last meeting)

AWB: how many people here have read these gists? (hands are raised) OK, about half the people... The others are going to have a hard time.

Slide 1, 2

Introductory

Slide 3

Main Issues

- @@create can expose uninitialized instances of built-in and host objects
- Necessitates numerous dynamic "is it initialized" checks in order to guarantee the invariants of such objects

AWB: The design long holding consensus was found to expose uninit instances. If it's an exotic or built-in with some invariant to maintain, instances may be produced that do not uphold the invariant.

- Concerns about the complexity introduced into the specification
- DOM APIs suddenly required to track
- Looking for a solution that doesn't expose the allocated but uninitialized instance.

Slide 4

Original Idea From Claude Pache

```js
class C extends B {
  constructor(...args) {
    /* 1: preliminary code that doesn't contain calls to a super-method */
    /* this in TDZ */
    /* 2: call to a super-constructor */ super(...whatever);
    /* this defined */
    /* 3: the rest of the code */
  }
}
```

- Added "receiver" argument to [[Construct]] that passes the constructor that `new` was originally applied to.
- Instead of pre-allocating `this` before entering the constructor, a TDZ exists until the super call

Slide 5

Addtional Idea Presented at Last Meeting

- `new*` token - Value is the "receiver" parameter from [[Construct]] or undefined if [[Call]]
- Can be used to discriminate "called as constructor" and "called as function"
- Provides access to original constructor for object intialization/intialization
  – `Object.create(new*.prototype);`

`new*` has been replaced by `new^`

- Value of `new^` is the "receiver" argument to [[Construct]]. Undefined if called as a function otherwise.
- `new^` chosen as alternative to `new*` since `new*` doesn't align with other uses of *, and new^ seems more appropriate.

Slide 7

`new super()`

- Use `new super()` rather than `super()` to "invoke superclass" constructor
  - `new super()` is always a [[Construct]] invocation
  - `super()`   is always a [[Call]] invocation

- Didn’t want to further confuse "called as a constructor" and "called as a function".
  – <id>() -- always means "called as function"
  – new <id>() – always means "called as constructor"
  – Even when <id> is `super`

Slide 8

`this = new super()`

- Original proposal had `this` in TDZ until explicit `super()` call. (now `new super()`)
  - Invisibly assigned to `this`
- Update proposal eliminates the implicit assignment by `new super()`.

- But allows an explicit assignment to `this`
  - Only in constructors
  - Only a single dynamic assignment
  - Subsequent assignments `throw ReferenceError`

MM: "Allows" explicit assignment to `this`, implies also allowed to call `new super()` without assigning to `this`?

AWB: Yes. Example is `Proxy(new super(), {...traps});`

Slide 9

`this = <expr>`

- RHS of `this` assignment in a constructor isn't limited to `new super();`
- May be any object valued expression:

```js
this = new super();
this = {x;1, y:2};
this = Object.setPrototypeOf([ ], new^.prototype);
this = new Proxy(new super(), handler);
```

Slide 10

Works in both class constructors and function constructors

```js
SubArray.__proto__ = Array;  SubArray.prototype = Object.create([].prototype);   function SubArray(...args) {
   if (!this^) this = new SubArray(...args);    else this = new super(...args);  }
```

MM: You can do assignment to `this` when called as a function?

AWB: No

Slide 11

Default object allocation (Base Classes)

- Class constructors without an `extends` and basic (function) constructors...
- ...Assign a new oridinary object to `this` if body does not have an explicit `this = `.
- These continue to mean the same thing:

```js
class Base {
  constructor(x) {
    this.x = x;
  }
}

function Base(x) {
  this.x = x;
}
```

Slide 12

Unqualified super references

- Until now ES6 has said that `super()` means the same thing as `super.<method name>()`
  - Implicit property access
  - Requires setup using toMethod (implicit or explicit)
- `super` in constructor needs to means "this constructor's [[Prototype]]", not "[[HomeObject]].prototype.constructor"
- It would be confusing if `super()` means something completely different in a constructor from what it means in an non-constructor method
  - Is this going to be used as a constructor or a method? function `f() {return super()};`

Slide 13

Eliminate unqualified `super` reference in non-constructor methods

```js
class Sub extends Base {
  foo() {
    super();     // now a syntax error
    super.foo(); // must say this instead
  }
}
```

- Unqualified `super` only allowed in class constructors and function definitions.
- Regular methods must qualify `super` references with a property access.

Slide 14

Default Value of `this` in derived constructors that don't assign to `this`

- Some alternatives
- `this = new super(); // super new with no arguments`
- `this = new super(...arguments); // siper new all args`
- `this = Object.create(new^.prototype); // oridinary obj`
- no value, `this` in TDZ at constructor start
- Most controversial part of design discussion

BE: Implicit object.create with new^ and subclassing a DOM base class, you end up with wrong thing

DD: Error better than not

(Domenic just named `new^` "new-hat")

AWB: Most preferred is Alternative 4.

Slide 15

The Winner: No Default `this`

- Eliminates issues of what arguments` to pass to implicit `new super()`
- Must assign to `this` in derived constructor before referencing it.

DH: In the constructor of a class that extends, no implicit this? Subclass would have to explicitly do something

AWB: Yes

Question about dead code removal, if `this = ` is in a dead code path

AWB: Tooling will have to be updated to understand new class semantics.

- Assignment to `this` in a function definition is meaningless, only works in a class body

MM: Any way to make derived constructor, base class and function rules?

```js
function f() {
  this = new super()
  this.x = 1;
}
```

MM: What happens is `f()` is called as a function?

AWB: If it includes `this = new super()`: Runtime error

AWB: `this = new super()` does a TDZ check

Questions about multiple `this = ...`

JM: In terms of dead code, linters, minifiers (any tooling) will have to become aware.

- Additionally, seems there could easily be incidental refactoring hazards (large constructors getting refactored into smaller ones, moving code around, accidentally breaking contextual auto-allocation).

AWB: Yes.

```js
// Throws because
class C {
  constructor() {
    this = undefined;
    return 5;
  }
}
```

MM: In the absence of a valid return, what would be the problem of returning the `this` value at the end of the constructor, even if the return value is a non-object?

BE/AWB: Consistency. This would be new semantics

MM: Sufficiently different, maybe not worth taking a chance on.

BE: I wanted allow return override. Guaranteed to return an object

AWB: If don't explicitly return an object, the `this` value is returned.

SM: Difference between assigning to `this` and returning an override?

AWB: None

YK: No TDZ?

BE: Invoke a function with new, the return is 5, not an error, just returns the original object

MM: comments about refactoring subclass constructor function to class constructor
```js
B.call(this) => this = new super();
```

(break)

Dave Herman and Yehuda Katz present counter proposal

atat.pdf

Slide 1

Introduction

Slide 2

Problem: Can create half baked objects.

Slide 3

(examples of Foo[@@create]())

Slide 4

Slide 5

- Uninitialized instances of builtin classes have to be implemented for every tupe in the entire web platform
- Uninitialized state

Slide 7

Solution: both allocator and constructor get arguments

Slide 8

```js
```

```
new C(x,  y,  z)

?

do {
  let obj = C[Symbol.create](x, y, z);
  obj[[Construc]](x, y, z);
}
```

Slide 9

- Builtins do all their work in the allocator.
- Constructors are noops.
- Impossible to observe uninitialized objects.
- Abstractable by WebIDL to avoid spec boilerplate.
- Abstractable by WebIDL implementations to avoid implementation boilerplate.

Slide 10

```js
Object[Symbol.create] = function() {
  return Object.create(this.prototype);
};

Array[Symbol.create] = function(...args) {
  let a = %CreateArray%(...args);
  Object.setPrototypeOf(a, this.prototype); return a;
};
```

Slide 11

```js
class Stack extends Array {
  top() {
```

~~~
    if (this.length === 0) {
      throw new Error("empty stack");
    }
    return this[this.length - 1];
  }
}

class Substack extends Stack {
  meep() {
    return "moop";
  }
}
```
~~~

Slide 12

~~~
```js
let PointType = new StructType({
  x: uint32,
  y: uint32
});

let ColorPointType = new StructType({
  x: uint32,
  y: uint32,
  color: string
});
```
~~~

Slide 13, 14

TODO: Copy slide

Slide 15

MM: Difference between the proposals:

- In new hat, case splitting on "if of new hat"
- Whereas, in ES6 and DH present, case splitting on @@create vs. constructor

BE/AWB/YK/DH: Agree.

BE: Clarify: this pertains to cases where you have split initialization and allocation, such as exotic objects, built-ins.

Discussion re: do not want to expose uninitialized stuff.

DH: issues with previous proposal, aka "new-hat"

1. new-hat syntax is there to distinguish call and construct
   - Agree that this is an issue to address in itself
   - There are ways to get this distinction without syntax
   - No way to wrap call and construct path because we're in declarative code
   - Too late for new syntax and specifically `new^` itself is not good enough to deliver to community.
2. Making `this` appear "mutable" (disagreements)
   - This is a bizarre change to the mental model of JavaScript
   - Appeared that all classes would _require_ assignment to `this`
3. Removal of unqualified super
4. In OO, it's considered an anti-pattern to use explicit conditions over method dispatch
5. Hijacking of the call path
   - It's reserved for subclasses
6. Have to do ALL of this to participate in subclassing

AWB: how does #5 differ from current? It's the same.

DH: Any reason why it can't call the construct behavior?

AWB: Construct allocates

DH: Not in this proposal

AWB: Haven't addressed what happens in constructor?

DH: (goes back to slide 8)

need more here...

AWB: Are you proposing that all initialization and allocation happen in @@create?

DH: no.

MM: When D inherits from B and you `new Derived()` that construct trap of Derived gets called. When Derived constructor calls `super()` does it invoke Base's call trap or construct trap?

DH: Need a way to distinguish new and call, but super or direct?

YK/BE: have to write another `if` if you want to define subclassable classes

DD: overriding the allocator is not part of creating a robust super class

YK: argues otherwise.

DD: enabling overriding the allocator of the subclass hasn't been a goal

BZ: in user code, needed in DOM

...continue.

DH: (explaining future friendly-ness of distinguished initialization and allocation)

Slide 16, 17

AWB: This is speculation of new forms?

YK: For illustration only. Speculate that you will have to build a constructor that defines all aspects of allocation and initialization, forced to figure out how the condition (or whatever userland pattern is)

- If we say the constructor is also the allocator, then we're boxed in and will have to deal with that.

DH: These speculated forms only illustrate goals.

- End up having to bless some protocol that distinguishes allocation and initialization.
- (wrap up, summarizing)

MM: Options?

1. Implement new-hat
2. Implement alternative @@create
3. Slip schedule?

3 => NO.

BZ: Want to see how these prevent uninitialized instances

AWB: (restating goals)

1. Self hosting
2. Subclassable built-ins and exotic object
3.

DH: Can prevent uninitialized instances from being observed with either proposal

RW: For exotic, built-ins and DOM, @@create produces the fully baked instance and constructor is just a no-op. For user code, that's uncommon.

...

BZ: I am trying to understand how you can guarantee initialization without exposing half-baked objects

DH: The answer is that you move the *strict invariant* initialization into `@@create`. To avoid repetition, you use trusted functions that are allowed access to partially initialized objects and encapsulation, and you only pass the `this` value from `@@create` to those trusted functions.

DH:

MM: Base class, derived class. The Derived clas is a client of the base class, and it's the responsibility of the Base class. It's often the case of the base class locking down property and that's OK.

BZ: Want to see some class that uses "magic".

- How does ColorPointType know how to

BZ: Does ColorPointType get to see an uninitialized PointType instance?

RW: No, because it overrode the one on Point

BZ: Wait so where does the initialization of ColorPointType and so forth take place?

DH: That's how struct types are defined.

RW: What about the examples we worked through last night.

YK: I think what's getting lost is that the normal way for people to use this is to ignore @@create.

BZ: The normal way is fine, I'm not worried about that.

YK: To be clear, what you're saying is fine, but I think it's important for everyone else in the room to know that the normal mode of operation has not changed.

AW: You're right but they also won't pass new^?

YK: But they will have created a non-subclassable object

ARB: I have a question. In the other design, there was a discussion about implicit calls to the superconstructor and whether we should pass the arguments. Both Marc and I made the strong argument that this was not a good idea. As far as I can see, it is doing the exact same thing, but even in a weirder sense, because you're passing the arguments through to the allocator even.

DH: That's true. Allocator signatures have to track constructor signatures. This means that if you are passing extra arguments that may be ignored --

?: Or used in a completely different way --

DH: right -- it means that you have to override @@create if you want there to be a difference

ARB: isn't that very common? in all cases where derived constructor changes what is passed to the base constructor

YK: not my experience at all. I think the common case is that you don't really change the arguments that much or just append arguments -- which happens to work ok thanks to JS semantics.

DD: I think what will happen is that people will say constructor is an anti-pattern, use @@create so that you only need to do one.

ARB: I've not used ES6 classes naturally but I find "transform" is the common case by far in other languages

DD: isn't that why you subclass -- to modify constructor?

<lots of people talking over each other>

MM: you have to have each level of @@create mirror the transform that occurs inside the constructor

YK: right

MM: do you have a pattern for that which is less onerous than the new^ pattern?

YK: I think that this will happen sometime but is less bad than the `if (new^)` pattern

AWB: No

YK: I guess we have to discuss whether we agree that people should design for

RW interjects: Here is the whiteboard example:

```
Function.prototype[@@create] = function() { return O.c(this.prototype); };

class A { }

class B extends A {
   [@@create]() {
    }
    constructor() {
    }
}

class G extends B {
    constructor() {
        ...
    }
}

var g = new G();
// because G has no @@create, we execute the @@create from B
// it all works out ok
```

RW: Follow:

```js
g -> new G() -> (no G[@@create]) -> B[@@create]() // done.
```

If B had no @@create

```js
g -> new G() -> (no G[@@create]) -> (now B[@@create]) -> (no A[@@create]) -> default //  done
```

DH: What Rick is trying to say is that for non-exotic objects, none of this matters

DH: Another way of putting this is that exotic objects are a bifurcation inherent to the space we're in, it's part of JS, and it becomes important to know whether there may be exotic objects involved

RW: Point I'm trying to make is that in the *most common case*, userland classes, we don't have to do anything different

JM: I can't just stay quiet anymore. I feel like we're overstating how common it is to subclass these kinds of "exotics"/built-ins

DH: I want to double down on this. Your prediction is wrong Domenic because I don't see people deciding that they should use the syntactically onerous pattern instead of just using constructor.

DD: I'm saying that based on Yehuda's argument, where he says to be robust you must be prepared for exotic objects, this design is superior, it doesn't seem like this design helps. Instead you just move things to @@create.

YK: That seems ok to me. To people who want to use exotic object subtyping, we can either say "here are a bunch of tools you can use to create a protocol", or we can tell them "here is something that works fine, but you have to use @@create"

DD: In that case, why constructor at all?

AWB: I'd like to explain why I walked away from this design. Fundamentally BZ's issue is about data-based invariants that must be established as part of the constructor, there will be a tendency to push those things into @@create. Besides being ugly, @@create has other downsides. Here's the real issue -- establishing these invariants, what does it mean? initialized private state. Let's think about the future. That method, the @@create method is a method on the constructor itself. That means to initialize the private state of the instance you have a function or method that is not attached to that instance that has ability to reach inside and get private state. So over here in the @@create method, however we might define it in the future, in the @@create method you can't say "this.@foo".

(people talk over each other)

MM: Not a capability leak -- if the @@create method is in the same class, it's fine.

YK: I agree but I think that's something we have to figure out generally for private state.

AWB: I'm saying that the model of putting private state into @@create, a foreign method and not the method of the instance, seems wrong.

DH: Here's the thing. @@create is not given an object to produce.

MM: I am confused about something Rick made. If you create a class G, just writing a constructor that calls super constructor, this is a broken class, because it didn't provide a parallel @@create that provided the same transform.

JM: In case where this is not extending an exotic class, it's ok. In the case of something native, like Date, it becomes important. My overall point is that I think it far less common to subclass Date than to subclass userland/non-native classes.

MM: But it's non-compositional.

DD: What's attractive about this = new super() is that it works regardless of whether it's a exotic class or not. Otherwise you have to know.

JM: I think you're right but we're taking what I perceive as an uncommon case and adding burden to the common case.

DD: People want to subclass Array a lot.

BE: Not common, not common.

JM: Let me avoid word "common" for time being. If we're talking about subclassing an ember view, that is not a builtin. It happens a lot.

DD interjects: But what if ember views want to upgrade to typed objects?

YK: It makes 'magic class' a part of the public API.

DH: Every class has two contracts. Contract to consumer and contract to subclasser. Magicness and exoticness is a thing subtypers have to care about regardless.

AWB: Dave, array is an exotic object. I can create a subclass of array today and not care whether it's an exotic object. So one step further. In ES6 as currently spec'd, I want to create a subclass of array, in the constructor I want to do something extra. I just want to log a message or something. I have to do something. In current ES6 design, I have to say super(), decide what arguments to pass, and do my extra code. But again I didn't have to think about whether Array was exotic not. It's only if I'm doing something that somehow interacts with the exoticness that I have to think about it.

BE: Domenic made a good point. I do think we have to argue about what's common though. Obviously there are piles of code building class hierarchy from plain objects -- but what if we want to go from an emberview to something using typed objects. Does one proposal require more changes than the other?

DD: No

NM: I disagree

...

YK: Answer with Ember subclassing proxies is that it is a semver incompatible change. I worked through this last night and I think it's ok.

DH: It's a drop-in replacement for *clients* but not subclassers.

YK: In the new proposal, it requires subtypers to type `this = new super()`, which they may not have had to do.

DD: I really like `this= new super()`, it's explicit, it makes more sense to me.

BE: But that's not the argument.

YK: I don't think we can call it a semver compatible change.

NM: Extensibility is a factor.

ARB: Generally true, right?

NM: Yes.

MM: I'm confused -- when I bring up the transforms of the arguments, response I got is that subclassing of exotics is relatively rare, which I agree, form of the argument is that it's rare enough that we shouldn't make regular code have to pay a price for the common case. Let's stipulate this is true -- under that argument, than the `if (new^)` goes away?

BE: But you still need `this = new super`, and other changes.

YK: But if you remove `if(new^)` you're making decision on behalf of others?

MM: If we apply the same counterargument, it doesn't need the `if (new^)`, all it needs more is `super()` becomes `this = new super()`

YK: I think you misunderstood the argument. I'm not saying you always have to write `if (new^)`. I'm saying that it is SOMETIMES true that you have to do it.

MM: I want to split the argument. In world 1, we are saying we don't want to burden normal code with worrying about subclassing builtins. For code that does not subclass builtins, it shouldn't need to worry about possibility that it might get reused as a subclass of a builtin. We should not need to pay the price.

MM: In that world, let's compare new^ to the proposal on the board. In new^, in world 1, if this is normal code, super() becomes `this = new super()`

YK: Yes but what this was saying is that IF YOU ARE subclasing something exotic, you need to do idiomatic things to get plausible results.

MM: But only on the hypothesized small amount of code that needs to subclass builtins.

YK: In the revived @@create, the burden falls on people are subclassing objects, and want to do a transform on the constructor.

BZ: And people who don't know if they want to subclass an exotic.

BE: I think they are lost anyway.

AWB: People who want to subclass to add methods, extend or override some behavior. You're not actually *changing* the exotic type, hence the constructor signature you want is basically the same as the normal class. So you have to define a custom @@create simply to transform the arguments.

ARV: I think that subclassing builtins will happen relatively frequently but it is VERY rare to then change it later.

YK: For this reason, I think people will forget to use the proper pattern, and thus screw their subtypes.

AWB: There is another approach. What we've done with @@create is split things into two halves. The way to solve this problem, doesn't have to be done at the spec level, one way is that you simply split out the initialization logic into a distinct method.

BE: I want to revisit. We are late in the game. I'm concerned we're hanging too much onto classes. I want to maybe push builtins into syntax or something else. Inventing all this stuff in the new^ proposal is a lot of new stuff, it's too late in schedule, ahead of implementation. DH's proposal hoists new object creation out of [[Construct]] which is also a change. Can we stick with ES6 and do what we want later?

DD: I see we have a few choices. Make implementors swallow @@create is designed, but implementors are very unhappy.

BE: Can we do things in the spec that do not create language level lock in?

MM: If we just do what's in draft ES6 there are too many observables because @@create is called by [[Construct]]?

MM: Lock in is that @@create makes uninit objects observable?

AWB: Nothing says an exotic object can't override it's construct internal method to not call @@create

YK: I think the thing to keep in mind that we want HTML Element to be subclassabe...

AWB: ...they can just keep overriding.

MM: What does Date do presently?

AWB: It invokes @@create and returns an uninitialized slot and blows up if you try to use its methods

MM: That is an observable state that user code can come to count on so it precludes future design that breaks such user code

DD: I think moving @@create into dom might work if you want to avoid locking it down at the language level

AWB: DOM objects can define a new create internal method. Their method can call @@create passing all the arguments.

BE: Marc's point about Date is interesting, though I believe Date can be done at user level. But since it throws, haven't we reserved right to say that we can change that in the future.

MM: I haven't had enough time to think this through. I really don't want to see us get locked in to what's in Draft ES6. What I'm trying to illustrate is that even to evaluate how bad lock in IS is something we can't do with confidence during this meeting.

AWB: Biggest lock-in that I see is that we lock in `super()` in a constructor. That's the path.

MM: Here are the 4 choices I see. Draft ES6. New^. Dave's proposal. Slip the schedule.

AWB: Or my @@create original design. Draft ES6 plus pass arguments to @@create.

MM: That has same issue with transform of constructor arguments, but it lacks having to rethink [[Construct]] path.

DD: I'd like to advocate for new^. I'd like to go through the objections and state why I don't those objections are good and I want to know the temperature of the room.

- `new^` as syntax: weird but rarely used
- `this = new super(...)` I think is great, it makes clear how JS works and how it's different from other languages
- hijacking call path -- I don't see a significant difference between writing two methods and writing an if statement
  - YK: There is no way to differentiate super call by subclass from client
  - DD: I don't see that as a big problem
  - YK: We're recasting it as "call constructor without allocation"
- requiring protocol creation between super/subclass types
- super.draw() --

BE: We should take a break. But we should not slip schedule. We can talk about the rest, but I want to emphasize that *we should be risk averse* and avoid locking ourselves into a bad path.

DH: the `this =` and the `new^` has a lot of machinery -- static analysis, if you say it is one thing and another. My point is not that it's absolutely wrong but it's a lot of risk.

BE: I think we're out of time for 6.

MM: I believe that if slipping is off the table, all the remaining choices are dangerous -- given that we have to make a decision, only one I would be willing is new^ or remove classes.

(general surprise)

BE: paths:

1. Drop classes (generally no)
2. Stay the course with draft ES6
3. new^
4. Dave & Yehuda proposal
5. DOM Overrides [[Construct]] to do its thing as necessary. Ugly. This is #2++
6. Remove @@create, built-ins do what ES5 did: Array subclass constructor was return override

- Cannot subclass built-ins

7. Pass args to @@create. This is #2++

MM: If doesn't find #2 acceptable, that includes #5, #7

NM: #2 should note that it exposes uninitialized state.

YK: Subclassing DOM, @@create returns dummy object, return new super

BT: least risk: punt on all this discussion and let people come up with subclassing protocols that meet needs

#6 results in syntactic lock-in. Devs will write `super()` and then we can't do `this = new super()` later.

MM: If we do #6 does it preclude having a future proposal with TDZ, unfulfilled this

DD/AWB: It would work if we do the "Domenic variation" in the future, with a default `this = Object.create(new^.prototype)` if there is no `this =`.

RW: #6 is a failure, we've told developers for years now that classes meant subclassing built-ins and we're taking it away.

DH: Unfortunately, implementors brought issues at the last minute.

BE: So we're talking about #5 then

- #3 is too dangerous this late in the game

AWB: Agree.

- #2, 5, 7 are minimal spec impact
- #6 big spec impact
- Crazy to do anything but variations of #2

BE: Motion... w/o judging `new^`, #3 is too much and risks the slip anyway

RW: Seconded.

MM: By same rule, #4

- User visible, uninitialized instances are too problematic, remove #2 & #5

BE: #6 remains, the developers suffer

AWB: Too much work

BE: #6 is out

YK: It's not clear that #2 is definitively broken, recall.

AWB: It's up to @@create to return a safe object

BE: BZ pointed out that there are places in the DOM where you want to alloc memory of a specific size. You don't want to alloc and re-alloc.

YK: There are already checks that exist.

BE: If you want to allocate the right size, you need the arguments.

AWB: All the types that are in ES6, that have size constraints are actually being dealt with in the spec.

BE: Mark also disliked #2 because you can call `B[@@create]()` and get back some blank object.

MM: A defensive class doesn't leak `this` during the construction process and a client can't get access and therefore no data dependent access

YK: Still have extra cost in addition checks

MM: Two brands per object

BE: Not going to work, will lose on performance

DD/YK/DH: (discussion of IDL)

EA: Most of the time DOM bindings don't have constructors

BE: What about the size constraints?

EA: Can work around it

AWB: The most common case is an oridinary object

MM: The zero brand and two brand cases...
- Constructor needs check the initialization brand

AWB: There is already logic in the spec that handles multiple stage initializations

ARB: Not convinced that #6 is off.
- Can we agree that #6 keeps all the other options open?

MM: I understand the workload issue, but what are the

ARB: `new^` is easier than @@create, could come sooner

RW: Can you explain?

ARB:

- all allocation codepaths have to change and do "virtual" calls
- adding plenty of checks for unitialized objects
- @@create is mutable, so every allocation gets deoptimized as soon as someone changes some @@create.
- can't easily be inlined

YK: I think they have to change for new^ also

ARB: no, allocation code path is neither decoupled nor exposed nor mutable under that proposal.

DH: Alternative design idea: treat the allocation as a proxy trap, with special hooks in to allow you to override it, e.g. in class syntax.

MM: 8. @@create -> [[Create]], no Symbol.create.

AWB: Instead of @@create method, have create slot on function objects and construct method calls it if it's there.

https://gist.github.com/bterlson/95ab741efa0bbd57f19d

MM/AWB: you could prevent it from being overriden in ES6

AWB: it's not a trap, it's private---perhaps private-inheritable---state of the function.

BE:

8. Constructor [[create]] slot

- own _not_ inherited
- no exposed to user code
- DEFER Reflect API called with args

ARB: Can mutate the prototype after the fact, so NOT inherited

DH: would implementers say that any time you subclass Array you get a deopt?

```js
Array[[Create]] = function (...args) {
    return %ArrayCreate%(this.prototype, ...args);
};
```

MM: I'm finding this quite attractive. Let's go back to the risk issue. If this turns out to work, how can we at this meeting gain enough confidence in it?

DH: biggest issue is any syntax for a custom allocator

AWB/BE: we should defer

DH: but note that there is no way we could do this without syntax in the meantime.

(Discussion of why 7 sucks and this new 8 is not bad.)

Key point: [[Create]] is NOT exposed or expressed in user ES6 code (*for now*).

DD: what about the observable uninitialized object via subclasses?

BE: [[Create]] is called before subclass constructor ever happens so you cannot in fact see the unobservable thing.

DH: Oh i see; we don't even need a minimal reflective API.

AWB: Note that, if you wanted to just define a function that was useable as a constructor but inherited/copied its create behavior from someone else, you could just do `class MyFuncWithCopiedCreate extends OrigFuncWithCreate {}`

MM: Does anyone object to #8

DH: Making sure, #8 is similar to #4 and more similar to #7. The only diff between 4 vs 7 is in 7: construct calls @@create internally; but in 4: it's hoisted out into @@create func.

ARB: Do uninitiliazed objects become unobservable?

AWB: ...since these things are closely coupled, it's not observable where you do it

ARB: Could remove init checks then

AWB: Checks aren't observable

MM: Or turn them into assertions

BE: Want design by contract in the spec

DH: Because you have no way to define own create slot, can't deal with arg transfers (up the create chain)

AWB: You're doing subclassing because you want the allocation from the builtin. So the way its defined is subclass constructor will have, as its create slot, the value of the create slot of Array (copied down)

AWB: When you say `new SubArray()`, you'll get Array create

DH: Yes, but you can't transform the constructor args passed from SubArray to Array

YK: Note that this only matters for args that affect the allocations

AWB: For ex: @@create for arrays don't care what the size of the arrays is

ARB: Problem: Now design of constructor signature is tied by internals of super class

BE: Yes, for ES6 that's true

DH: This is an inherent part of JS. Distinction between exotic types/normal types. When subclassing exotic types, you must participate in the parent's protocol. You have to know more things in these case (constructor signatures, create sigs, etc)

DD: Dislike that...

BE: Can you get it later [post ES6]? Probably

: Can you do super.apply?

BE: Depends.

: Do it in the TDZ

DH: Any new syntax is too high risk for ES6. #8 while most conservative, it introduces the transform problem without a solution

MM: Can someone clarify that we can get out of the scenario in the future

DH: As long as we have a way in the future of doing something like:

```js
class Stack extends Array {
new() { // overrides the create -- strawman syntax only!!
return [];
}
constructor() { ... }
}
```

DH: Gives a way to transform the args

MM: If transforming args up construct chain, need a parallel transformation up the 'new' chain?

DH: Yes

MM: The burden of doing the transformation twice is too awkward.

DH: No question there's a cost. Seems livable to me, but...

BE: What's alternative?

AWB: There's kind of a workaround: Proxy. Proxy overrides construct. Could filter args in construct method of a proxy.

DH: Imagine mixin that you extend from

MM: Satisfied it's possible

AWB: I'm satisfied it's *probably* possible -- not sure yet though

DD: Should talk about promises. Specifically about subclassing: I think if we moved everything into @@create, Promise methods would call this.@@create instead of this.constructor, then could have own custom subclass constructors

BT: No because .then() ...?? base class blocks all subclasses

BE: There are circumstances where you walk the subclass hierarchy to base class

[discussion about promise implications]

DD: I haven't thought about it enough to really say. I don't like the proxy thing, but...

YK: Could imagine a thing that works like a proxy, but isn't actually a proxy

BE: A mixin seems good

DD: Shame that we, by default, need this hierarchy.

MM: Let me try out a scenario that DD and I will find attractive: If we start with #8, then after ES6 want to get to new^, if syntactic occurrence of "new super" then we don't call our local create, and therefore there's no prob with args transform

MM: I think that just works

YK: All I would say is that #8 seems future proof to that

MM: Yes.

DD: That's good, I still like new^ and would be good option for ES7

AWB: Would be helpful if proponents of new^, as we update the spec, looked at the spec and made sure it works (for the future)

DD: In the world of #8, user code that wants to call super class just does super()?

AWB: Yes

MM: With that issue resolved, anyone object to declaring concensus on #8?

ARB: Should we sleep on it?

RW: Lets record it, then immediately decide on it in the morning first thing.

[lots of head nods and agreement mumbles]

BE: Gotta revisit trailing commas from yest. I don't think there are any grammar problems, right?
BE: I don't know, I don't care.

AWB: I guess they're ok, but they have a smell

DH: One clarification, is there already a create trap? Are we talking about adding one?

AWB: No. There's a construct only. Not going to add one

DH: I don't understand the use-a-proxy idea as a solution

AWB: Construct trap gets the args, it transforms

DH: Obviously sub-optimal, but at least expressable and we can work on it
DH: [repeats] Would be nice to have good syntax eventually, but good for now.

#### Conclusion/Resolution

- #8
- https://gist.github.com/bterlson/1fe0b0dc0ef3e71ff6e3

## 5.1 Trailing Commas in Function Call Expressions and Declarations

(Jeff Morrison)

trailing_comma_proposal.pdf

JM: Review...

ARB: Supported for consistency

RW: If people don’t want them: use JSHint. No objections otherwise

#### Conclusion/Resolution

- Stage 1 acceptance

## RegExp Globals

DD: RegExp globals reform recap.

MM: Can we test on deleted properties?

BE: What about a flag?

- Per instance
- Flag to be determined
- Specified in Annex B

#### Conclusion/Resolution

- Stage 0 acceptance

# September 25 2014 Meeting Notes

Brian Terlson (BT), Allen Wirfs-Brock (AWB), John Neumann (JN), Rick Waldron (RW), Eric Ferraiuolo (EF), Jeff Morrison (JM), Jonathan Turner (JT), Sebastian Markbage (SM), Istvan Sebestyen (phone) (IS), Erik Arvidsson (EA), Brendan Eich (BE), Domenic Denicola (DD), Peter Jensen (PJ), Eric Toth (ET), Yehuda Katz (YK), Dave Herman (DH), Brendan Eich (BE), Simon Kaegi (SK), Boris Zbarsky (BZ), Andreas Rossberg (ARB), Caridy Patino (CP), Niko Matsakis (NM), Mark Miller (MM), Matt Miller (MMR), Jaswanth Sreeram (JS)

## 5.8 Object Rest Destructuring and Spread Properties

(Sebastian Markbage)

Spec https://github.com/sebmarkbage/ecmascript-rest-spread

Request Slides

SM: Update: own properties?
- Need to be own properties

```js
Object.prototype.hostile = 1;

let { ...o } = {};

let o = { ...{} };

o.hostile; // 1
o.hasOwnProperty("hostile"); // true
```

MM: clarifies hostile vs. accidental

MM: When does Object.assign do toMethod?

RW: Never. That was designed for Object.define/mixin

AWB: Confirm

SM: Mental model:

```js
...o

expands keys = Object.keys(o) to
o[keys[0]], o[keys[1]], o[keys[2]]
```

Security Consideration?

Syntax introduces a new way to determine "ownness" without going through (patchable) library functions:

- Object.prototype.hasOwnProperty
- Object.keys

MM: Explains that SES is capable of patching the above by replacing these APIs. Rewriting syntax is undesirable.

Discussion about ownness and enumerability

YK:

MM: If we proceed assuming weak maps are not slow, then they won't be (i.e., browsers will finally switch to the transposed representation)

YK: As a lib impl, weak maps are slow and I have no reason to believe they'll be fast in the near future, so I won't use them

AWB: Not our job to design lang around what things are slow today

ARB: concerned about proliferating the use of enumerability

AWB: Had this discussion many times. Enumerable is obsolete reflection of semantics of for-in. Don't want people to use enumerable to start meaning new things.

YK: We agreed it's obsolete function of for-in, but it is widely used. So can't change the way it's assumed to work

AWB: Today, in es5, enumerability used as a way to deal with copying

MM: WRT proposal on table, it is a copying API. So you're agreeing with the fact that the proposal on table is sensitive to enumerability

YK: Yes

ARB: This is only really useful for record copying

AWB: Let's talk about own again. This is an extension of obj destructuring -- which, as it exists today, does not restrict prop accesses to own properties.

AWB: However, it does restrict/ignores enumerability

AWB: Implication of this being own: You couldn't use a tree structure to represent a set of values:

```js
BaseOpts = {__proto__: null, o1: 1, o2: 2, o3: 3, ... oN: n};
ThisTime = {__proto__: BaseOpts, o7: 14};
foo(ThisTime);
```

AWB: Now, inside of Foo...

AWB: Since you're excluding non-enumerables, normally things in Object.prototype are excluded. So not worried about picking up those things

YK: You're worried about picking up someone's old prototype extensions. It's true people could do that, but in practice people don't

MM: We have several arguments that say own is what's expected

MM: In ES3, assumption is there's all sorts of gunk on Object.prototype because no way to get rid of it. Reason it was restricted to own-ness was because there was a desire to not iterate stuff on Object.prototype

YK: There's a notion of copying today that means "non-enum, own"

AWB: Notions in JS today are diff from notions 5 years ago and 5 years from now

MM: We can't accommodate legacy in this way forever. We're in a history dependent trap, we should make this (enumerable + own)?[verify]

AWB: How does this play with/needed in the context of other extensions? For example, record types vs property bags to represent options. If people use typed objects vs regular, would expectations change for this syntax?

SM: Brings up another proposal: Records (new proposal) in same realm as TypedObjects, but simpler syntactically

AWB: In the past, when focusing in on a microfeature, it makes sense. But when looking at features more broadly those proposals make sense differently. There are enough things that are coming soon that need to be considered here as well

MM: This feature is on the same "future table" as those other things, so they'll be considered together as we move forward.

SM: Record types don't have a concept of prototype chain, so not even a consideration; So this should operate consistently between those and regular objects

SM: (tries to move on to a new slide)

ARB: so ownness is clear, but are we settled on enumerability?

MM: yes

SM: I think it should be settled, and any argument against the enumerability policy here also applies to Object.assign

YK: Object.assign is meant to be a widely-used mechanism for copying

ARB: I see your point, but am still concerned...

YK: I think the problem is that people don't like enumerability. I don't like enumerability. But enumerability is how you design copying behavior in this language.

AWB: Is this feature valuable enough to make this as syntax rather than something that lives in a library

MM: Need a functional mechanism for updating records

AWB: This seems like a nice feature, but not sure why this should make it in over other features.

YK: Symmetrical with destructuring, so easy to understand.

MM: This seems like a smaller burden on programmer psychology

AWB: Well, it is a burden. Spread and rest used to be an enumerable list of things, and now we have ... mean something else.

MM: There is a cognitive cost, but because of the analogy it's much less than a new feature.

MM: When we previously produced large specifications (through ES6) we used to determine the complexity everything together. Possible that the yearly release will make it more difficult to budget for complexity.

DH: Also makes it harder to say no to a feature that makes sense locally and has gone through the process.

MM: We should allow for something to get rejected even after it has gotten through all the stages.

YK: It's an implicit requirement. Should make it an explicit requirement.

MM: Agree.

DH: Recognize that this feature is fitting in an existing syntactic space and is rounding out a syntax that already exists.

ARB: It's subtly different...

DH: You'll have to learn it

```
function ComponentChild({ isFoo, ...rest }) {
  var type = isFoo ? 'foo' : 'bar';
  return ComponentBase({ ... rest, type });
}
```

AWB: What happens if rest is an array?

MM/SM: It creates the enumerable array properties.

AWB: But we use iterators.

MM: It should be dependent on the syntactic thing containing the .... In an object literal, ... will enumerate properties.

SM: Stage 1?

#### Conclusion/Resolution

- Stage 1 approval

AWB: Be sure to mark agenda items that want to advance with some kind of notation, this will help to get pre-meeting attention.

(Confirmed by all)

RW: Use the rocketship icon

YK: I feel like the rocketship icon should be for proposals which are ready to launch

DD: If you bikeshed on the rocketship icon I will change it to a bikeshed icon.

RW: :D

## Loader pipeline

AWB: Working on module spec. Questions: Loader pipeline. Can we simplify modules for ES spec?

"A.js"
```js
export let g = 5;
import f from "B.js";
f(g);
```

"B.js"
```js
import g from "A.js";
export function f(z) {
   return z * g;
};
```

AWB: Essential in ES6 that the semantics of above example code is well defined.
 - None of it depends on how the loader is parameterized.
 - Strictly at the declarative level of the language

BE: So what goes on the ES side?

AWB: Syntax and semantics (static, linking, and runtime) of declarative modules. Linking does have to be there, but it's linking at the declarative level of the lang.

DH: So you have some small number of low-level integration hooks in ES that expose enough for browser implementors design and build loader pipeline themselves?

AWB: They're at the same level as the other host hooks that we have

AWB: The only host hook is the one that says "there's a request for a module name (from referrer), give me the source code"

DH: To clarify: I think what Allen means by "hook" is not user-visible, it's visible to an engine. It's a spec device used to factor out the pipeline. It's not available to user code, just to the internal semantics of the pipeline.

MM: And we can move the pipeline into a separate spec.

EF: "Authoring and runtime"

- Authoring: actual writing of modules
- Runtime: the loader

(Discussion about Loader polyfill)

(Discussion of how the loader spec would be a separate document)

YK: The loader pipeline will be done in a "living spec" (a la HTML5) so that Node and the browser can collaborate on shared needs.

#### Conclusion/Resolution

Loader pipeline goes into a separate spec: living document that deals with integration

AWB: Retitle? ECMAScript 2015 (6th edition), and so on

## Train Schedule

DH: Let's define the schedule

YK: ES2015 is the first train.

## Types
(Jonathan Turner)

type_annotations.pdf

JT: Goals
- Short term
- Reserve syntax used by TypeScript, Flow, etc. for some form of annotation
- Venue for collaboration among interested committee members
- Long term
- Consensus on a shared syntax fr many varied type annotation implementations
- Consensus on

- Additionally, a shared syntax for interface definition for documenting API boundaries (.d.ts files)

Examples & Demo

...

Rationale: Why Type Annotations?

- Toolability
- Closure
- TypeScript
- Flow
- JSDoc
- Performance
- Asm.js
- Hidden classes/runtime inferences
- API specification

- DefinitelyTyped/.d.ts
- WebIDL
- JSDoc
- Runtime checks/Guarantees
- Guards
- Contracts

Rationale: Why Standardize?

copy from slides

JT: Looking for Stage 0 blessing to pursue type annotations a la TypeScript and .d.ts file definitions.

MM: You've presented annotation, what about checking?

JT: Type checking not defined

- mention of python type design

MM: I'm not familiar

JT: TS, Flow have different type checking rules that will hopefully emerge

DD: Draws comparison to divergent promise implementations that were successfully unified

ARB: the result is rather terrible

MM: The fact that `.then` came together was a miracle and shouldn't be a practice.

DH: If we want to agree on reserved syntax that currently has no legal JavaScript overlap, that is it fails, not ignored. If such a thing can be agreed on, then different groups can develop around the syntax divergently. Cannot expect to reserve _behavior_. The rationale slide is far too vague.

- Goal: TS and Flow are taking a risk where TC39 could easily standardize on a syntax that invalidates those project's uses

JM: no attempt to standardize the entire syntax

- Python: imagine similar approach

- annotations
- arbitrary expressions evaluated then attached to the function object
- language doesn't need to use them in runtime
- tooling may use them
- allows both static and runtime tooling

- (this point was about third party consumption of type information, needs to be filled in)
- no guarantee that TS or Flow won't continue extending the grammar

YK: If you give type checks anything but an error, you can't create different semantics later

DH/YK/DD: mixed discussion re: history of type design sets constraints

JM: We need to start making these things possible by putting the capability in the language

STH: underestimating the complexity (refers to DH work)
- Types seem to be doing well in their current form (compile to JS)
- what is the problem we're trying to solve here?
- RE: Python, the underspecification is already causing problems w/ competing groups

JM: to move forward we really need a space that's reserved

DD: Wouldn't have proposed this as types, it's closer to parameter types and returns

YK: decorators suffered the same syntactic space arguments.

MM: I'd propose that you enter Stage 0 with TypeScript.
- TS made a choice and proved that this choice has utility.
-

DH: Don't need TC39

JM: How do we know when it's time to come to TC39? We need TC39 to help with progress

JM: Various projects working toward similar goals: TypeScript, Safe TypeScript, Closure Compiler, IBM has a project, Flow. Ongoing research that should be collaborating and coordinating with TC39

BE: codify annotation grammar, build object that you can reflect on. Not ready to do that.
- Make reserved syntax, we can do that.

MM: It's just a 16.1 restriction on extensions.

BE: ECMAScript allows and always has allowed implementers to make syntax extensions.

YK: Nashorn adds "#" comment

DH: Difference between TS, Flow and Nashorn

RW: (couldn't say outloud) Nashorn changed JS syntax; TS, Flow compile to JS

JT: Just want to reserve/restrict certain basic block of syntax that projects can use

DH: Syntax reservation has value. Attempting to define some minimal semantics is a bad idea.

BE: (copy grammar from whiteboard)

JM: Would like to be able to point to document for this

AWB: Some kind of "statement of future direction" document

RW: Similar to Future Reserved Word, it's a "Future Reserved Grammar/Syntax"

(agreement)

AWB: (explanation of how this could work and documented)

SM: Concrete spec proposal and what goes into next release?

- Will need to converge eventually
- Can do it on our own
- Could do this as part of the TC39 process

RW: suggest proposal Future Reserved Grammar doc for next meeting to ask for Stage 0

(discussion about responsibility)

JM: Seems like "type systems of some kind" have interest. Start conservatively, Future Reserved Grammar/Syntax etc, and build from there.

JT: Stage -1: reserved grammar

DH: Stage 0 for this:

- Reserve syntax via Future Reserved Grammar/Syntax
- Does not compute
- Is an error
- cannot ever create an incompatible change

AWB: Make a motion that TC39 is creating an area of research in types and type annotations and all members are welcome to get involved?

DH: As long as we maintain balance and prioritize.

BE: Concern about opening the door too wide.

SK: What about work on extensions that _require_ semantics?

ARB: you can't know what type syntax you need without knowing the semantics. In particular, Python's type syntax as just expression syntax doesn't scale, you generally need different constructs on both levels

BE: for example generic brackets

DH: What is the grammar?

JM: (python expression example)

What's the conclusion?

ARB: make colon syntax reserved

RW: there's more to it!

#### Conclusion/Resolution

- Create Future Reserved Syntax (extension restrictions)
- Syntax error
- Define `a: T<U>`

## 5.10 global.asap for enqueuing a microtask

(Domenic Denicola and Brian Terlson)

DD: Want enqueue microtask, which is capable of starving the eventloop

AWB: As spec writer, I don't know what this is

YK: In JS there is a frame, it loops

MM: Is the queue, the same queue that promises queue into?

DD: Yes

YK: Want a way to queue a job that's guaranteed to run before all other tasks

AWB: There are spec mechanisms that define ways to create a job in the queue

DD: Don't care what it's called just want it to express the intent, which is faster than setImmediate

Discussion about the semantics and defining the order of execution. MM is objecting

YK: A non-normative note?

DD: No, if it's non-normative I don't care, I want it normative

Issues about host interference with expected run-to-completion model

AWB: present job and job queue mechanism intended to describe the two things we needed to describe and knew there would elaboration. Go ahead and develop a proposal.

YK: Concerned that explanation problems lie in using browser terminology

- Micro task is part of run to completion

- Task queue is not

AWB: jobs run to completion

MM: Job queues are always async by definition

- We have terminology, please use the correct terminology

YK: Ok, won't use "synchronous"

MM; multiple queues in a priority mode

- You want the microtask queue to have a strictly higher priority

- We may even specify priority queues

DD: Want to specify `global.asap`

- Accepts a function

- Enqueues in a high priority queue

DH: Think this is awesome

- We need a generic model for job scheduling.

#### Conclusion/Resolution

- Stage 0:
  - Some way to publish into a queue
  - priority queueing