Minutes for the: ***36th meeting of Ecma TC39***

in: ***Boston, MA, USA***

on: ***17-19 September 2013***

1 Opening, welcome and roll call

1.1 Opening of the meeting (Mr. Neumann)

Mr. Neumann has welcomed the delegates at the Bocoup, in Boston, MA, USA.

Companies / organizations in attendance:

Mozilla, Google, Microsoft, eBay, jQuery, Yahoo, Adobe, Apple, IBM, Free University of Brussels, Facebook (guest), Indiana University (guest), Lab49 (guest)

1.2 Introduction of attendees

John Neumann – Ecma International

Istvan Sebestyen – Ecma International

Mark Miller – Google (part-time, phone)

Douglas Crockford – eBay

Luke Hoban – Microsoft

Oliver Hunt (part-time, phone, Apple)

Tom van Custen (part-time, phone, Free University of Brussels)

Simon Kaegi (part-time, phone, IBM)

Philippe le Hageret (W3C / MIT)

Allen Wirfs-Brock – Mozilla

Anne v. Kesteren – Mozilla

Erik Arvidsson – Google

Waldemar Horwat – Google

Andreas Rossberg – Google

Domenic Denicola – Lab49 (guest)

Sam Tobin-Hochstadt – (part-time, phone, guest. Indiana University)

Alex Russell – Google

Eric Ferraiuolo – Yahoo!

Reid Burke – Yahoo

Matt Sweeney – Yahoo!

Rick Waldron – jQuery

Dave Herman – Mozilla

Brendan Eich – Mozilla

Rafael Weinstein – Google

Boris Zbarsky – Mozilla

Nicholas Malsakic – Mozilla

Tim Disney – Mozilla

Dimitry Lomov - Google

Jeff Morrison – Facebook

Sebastian Markbage - Facebook

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

Ecma/TC39/2013/045	Results of the postal ballot regarding draft Standard "The JSON Data Interchange Format" 1st edition
Ecma/TC39/2013/046	Ecma Workshop on Dart
Ecma/TC39/2013/047	Minutes of the 35th meeting of TC39, Redmond, July 2013
Ecma/TC39/2013/048	Agenda for the 36th meeting of TC39, Boston, September 2013
Ecma/TC39/2013/049	Eleventh draft Standard ECMA-262 6th edition, September 2013
Ecma/TC39/2013/050	Results of GA postal vote (The JSON Data Interchange Format)
Ecma/TC39/2013/051	TC39 RF TG form signed by Microsoft
Ecma/TC39/2013/052	Draft Standard ECMA-262 6th edition, Rev. 19, September 27

2 Adoption of the agenda (2013/048)

Agenda for the: 36th meeting of Ecma TC39 (final version on Github)

```
in: Boston, Massachusetts, USA
on: 17 - 19 Sept. 2013
TIME: 10:00 till 17:00 EDT on 17th and 18th of Sept. 2013
      10:00 till 16:00 EDT on 19th of Sept. 2013
LOCATION:
    Bocoup
    355 Congress St.
    2nd Floor
    Boston MA, 02210
    http://goo.gl/dmwn3
Mr. Rick Waldron's email: waldron.rick@gmail.com
Mr. Rick Waldron's local phone number: 857-540-9264
Nearby hotels: http://goo.gl/AaK6L
```

Please register before 10th of Sept. 2013.

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 (2013/048) – approved by TC39
3. Approval of the minutes from July 2013 (2013/047) – approved by TC39
4. ECMA-262 6th Edition
    i. Arrow functions empty parameter list elision. (Brendan)
    ii. Reconsider decision to make Typed Arrays non-extensible: http://esdiscuss.org/topic/non-extensibility-of-typed-arrays
    iii. Math.hypot (dherman)
    iv. symbols (dherman)
    v. modules status report (dherman)
    vi. unbound variables (dherman)
    vii. JSON.stringify and unpaired surrogates: https://mail.mozilla.org/pipermail/es-discuss/2013-September/033293.html
5. ECMA-262 6th Edition Open issues discussion. (Allen)
    i. splice, cross-realm Arrays, and subclassing: http://esdiscuss.org/topic/array-prototype-slice-web-compat-issue
    ii. evaluation order and the [[Invoke]] MOP operation: http://esdiscuss.org/topic/evaluation-order-and-the-invoke-mop-operation Also conditional [[Invoke]] issues
    iii. @@unscopable property: access once per with or once per lookup? http://esdiscuss.org/topic/unscopeable
    iv. should @@unscopable arrays defined in standard be frozen?
    v. Math.roundFloat32 --> Math.fround (other possibilities: f32Round, roundF32)
    vi. Backwards compatability, Unicode UTR31, and ES identifiers (U2e2f) http://esdiscuss.org/topic/backwards-compatibility-and-u-2e2f-in-identifier-s
    vii. note in 11.6 WRT Unicode versions, update to Unicode 5.1? http://people.mozilla.org/~jorendorff/es6-draft.html#sec-11.6
    viii. line terminators in template strings. Should they be normalized? (see note in 11.8.6)
    ix. 12.2.4 note says we decided (Jan 2012) tail calls only in strict mode. Is this still correct ???
    x. Function parameter scoping and instantiation (http://people.mozilla.org/~jorendorff/es6-draft.html#sec-9.1.16.11)

xi. Disallow? let undefined; const undefined; class undefined {}; module undefined from "foo";

6. Post ES6 Spec Process. (Rafael)
7. Object.observe status report. (Rafael)
8. data parallelism API (dherman, nmatsakis)
9. Promises: https://github.com/domenic/promises-unwrapping (please review ASAP, cannot prevent shipping much longer, Anne)
10. Status Report ECMA 402
11. Status Report Test 262
12. Report from the Ecma Secretariat
13. Date and place of the next meeting(s)
    - November 19 – 21, 2013 (PayPal - San Jose)
14. Closure

The agenda was approved.

3 Status Reports

3.1 Report from Geneva

3.1.1 Brief report about move to a Tc39 RF TG

Mr. Sebestyen said that the IPR ad hoc group is still working out a solution for a software contribution from non members, but the IPR ad hoc group has not finished their work. So the external contribution is not yet for software. This will be the next step.

Regarding Registrations to the TC39 RF TG the Ecma Secretariat has only received a very few registrations yet. So, the current modus of operation is still RAND. He urged that TC39 members if interested should officially register to the TC39 RF TG.

3.1.2 Ecma recognition program

Mr. David Fugate has received the Ecma Recognition award and the award has been dispatched.

3.2 Json

After successful TC voting the JSON ballot is ongoing. The deadline for the GA voting is October 6, 2013. The JSON standard will be ECMA-404

4 Date and place of the next meeting(s)

Schedule 2013 meetings

- November 19 – 21, 2013 (PayPal - San Jose)

5 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** for taking the technical notes of the meeting.

Annex 1

Technical Notes (by Rick Waldron):

From: Rick Waldron [waldron.rick@gmail.com]

Sent: Wednesday, September 25, 2013 4:47 PM

To: TC39; John Neumann; Patrick Charollais; Istvan Sebestyen

Subject: Updated: September 2013 Notes

These notes include additions and corrections and supersede the previously submitted notes.

Sept 17 Meeting Notes

John Neumann (JN), Dave Herman (DH), Istvan Sebestyen (IS), Alex Russell (AR), Allen Wirfs-Brock (AWB), Erik Arvidsson (EA), Eric Ferraiuolo (EF), Doug Crockford (DC), Luke Hoban (LH), Anne van Kesteren (AVK), Brendan Eich (BE), Brian Terlson (BT), Rick Waldron (RW), Waldemar Horwat (WH), Rafael Weinstein (RWS), Boris Zbarsky (BZ), Domenic Denicola (DD), Tim Disney (TD), Niko Matsakis (NM), Jeff Morrrison (JM), Sebastian Markbage (SM), Oliver Hunt (OH), Sam Tobin-Hochstadt (STH), Dmitry Lomov (DL), Andreas Rossberg (ARB), Reid Burke (RB)

Welcome

RW: ...Logistics...

Agenda

Promises to be discussed Thurs. PLH from W3C will be here on Wed/Thurs.

BE: won't be here thurs, and that suits me.

JN: objections to using this agenda?

(crickets>

JN: next issue is approving the agenda.

Approved.

JN: Minutes approval?

RW: (Confirms no changes since last review)

Approved.

JN: have a strong duty to surface the agenda to ECMA and right now we don't know how to do that. Want to publish something to ECMA 3 weeks before the meeting and one week before, a final version of the agenda.

AVK: how about we give them a URL?

IS: the issue is that we haven't had a fleshed-out agenda 3 weeks before the meeting. There hasn't been anything there.

AR: this seems like a mismatch between the historical timings of meetings and our accelerated cadence.

DH: We talked about this in last meeting, where we agreed that everything would be in the agenda one week prior. If we don't accept new items, the agenda is a dead letter.

AR: we can commit to having a skeleton that has stuff we will usually talk about, but not more.

RW: in the past, an email went out that helped remind people. If we sent out a reminder email at the halfway point, it might help people remember to add agenda items then.

DH: Let's strategize which items are best to work on and which day.

- Modules: Wait for Andreas

- Typed Arrays: Wait for Dmitry, Oliver

- Symbols: wait for Andreas

- Proxies: wait for Tom VC

DH: Arrows, Math.hypot today

EA: Spec process

DH: We can do this out of order.

...Let's discuss Data parallelism while Niko Matsakis is present.

11. Status Report 262

AWB: I'd like feedback and comments on the new re-orderings.

DH: I know that Jason Orendorff has some issues (es-discuss: https://mail.mozilla.org/pipermail/es-discuss/2013-September/033314.html (starting with Olliver Hunt))

AWB: New sections...

- Runtime Abstractions
- Group Of Chapters that define the language
- Lexical Grammars
- Syntactic Grammars

AWB: There was some recent discussion about how you track down semantics when you have so many named semantic algorighms that are associated with productions. I've done work subsequent to r18 that adds a "see also" section at the top of each segment so you can

BE: what do people think of the new organization?

(general approval>

WH: The latest version sent to the ECMA document repository is from May. Why aren't new versions being sent to the ECMA repository?

(group directs WH to the latest version not in the ECMA document repository>

(googlers + facebookers arrive>

BE: wait there's more Googlers? We have to bring more Mozillans now... (I kid, I kid.)

AWB: At the last meeting we sucessfuly deluded ourselves into thinking we didn't need NoIn productions. We've been unconvinced, so I reintroduced them.

WH: I've been looking into this NoIn business

AWB: OK, well if you come up with a solution for it!

WH: I got my grammar verifier up and running again.

DH: Ollie spent a long time looking at them and didn't come up with a solution.

AWB: naked yield is supported. The assignment part is optional.

EA: for yeild*, the expression is needed.

DH: agreed

- yield * wouldn't make sense without

AWB: WH brought up that you can't have a conditional reserved word, so what I did is ensure that yield is always a reserved word which, in contexts where it can be an identifier is treated as such.

DH: contextual reserved would would be another instance of feedback from parser back to lexer (a la slash); Allen's approach is more elegant.

BE: yes, it's the right tradeoff.

AWB: overall status: thanks to Dave, in the current working draft we're starting to bring in module grammar productions and semantics.

- module grammar productions
- module semantics
- Confident in the progress

BE: we have a separate agenda item on this, yes?

(yes>

AWB: still hanging out there and need work

- no work done on unicode regexp
- no work done on updating regexp to web reality

LH: Last consensus on "web reality" was that it would be in an annex ...

LH: there's text (http://wiki.ecmascript.org/doku.php?id=strawman:match_web_reality_spec) that patches some rules.

LH: the reason we got hung up is we thought there might be some grander unification with the regexp unicode flag, and I don't know what the status is on that.

LH: we should integrate the web reality piece into the spec.

AWB: do we want to defer the Unicode part

BE: did we have someone identified to work on it?

(norbert>

AVK: I thought there was discussion that agreed what was in and what was out?

BE: If Norbert doesn't do it, it's not going to happen for ES6

WH: (Q about the yield)
- Will allow yield as an identifier?

AWB: Yes

WH: This will break

(whiteboard)
```js
yield * 1;
```

is that a multiplication or `yield *`?

AWB:
(whiteboard)

YieldExpression := [inside a generator] yield aaugh this keeps getting erased

AWB: inside a generator function, "yield" is a keyword that introduces a generator expression.

WH: inside a generator, this ("/") will be a RE, outside it will be a division symbol?

WH:

```js
yield /a/g;
```

is this yield divided by a? or yield a regexp?

BE: so this is like the regular expression delimiter, it's going to need to have feedback from the parser to the lexer.

?: If you're inside a generator, it's lexed as a regexp. If not, it's lexed as division.

DH: I think that isn't affected here, the lexer is always fed the parsing context; it's not an ambiguity because it's determined by context.

WH: What's "inside a generator"? What if you're inside, say, an arrow function inside a generator?

?: Then you're not actually inside a generator.

WH: But you need to lex the thing to parse the thing to find out that you're inside something nested inside a generator, so that's begging the question.

BE: split grammar at higher level, InGenerator, NotInGenerator, something something something.

BE & DH: use new tricks, e.g. parametric productions. Even NoIn, back in the day, we would have preferred to use those.

AWB: Yeah, if we have to do this for yield, I'd be happy to switch to parametric productions, since it avoids having four forms of the expression grammar.

BE: alternatives is to make generator bodies strict.

DH: We shouldn't make the users life worse just to make the spec easier.

WH: well we are prioritizing the user, think of syntax-coloring editors, they also now need to know these complicated rules, too complicated and they'll get them wrong.

DH: in practice editors get things wrong

BE: and then you can pull request against them so it's all good.

DH: new implicit modes will bite more often

BE: let and yield are reserved in strict mode since ES5

DH: Parameterized productions are the cleanest solution

BE: Much better to parameterize then to duplicate productions.

BE & AWB: let's do a smaller group on parametrizing the productions.

...

AWB: Is there a point on binary data?

- Don't think we're going to get them in

BE: related to spec process, getting things done painless and in a rapid-release fashion. let's talk separately.

AWB: Status wise: no progress on Binary data Typed Objects proposal with regard to the spec.

RW: (clarified binary data vs "typed objects")

DH: binary data name is bad, "typed objects" better. Because typed objects as specced are not actually good for binary data.

Typed Objects (formerly known as binary data)

AWB: typed arrays and dataviews are in ES6 spec, "structs" are not.

DH: Dmitry *is* working on this, it's not 2014 yet, we shouldn't definitively say it's not going in.

AWB: well, I'm trying to manage expectations.

BE: Let's revisit after we get a little further with the other items.

4.1 Arrow Functions

BE: When I proposed Arrow Functions, the empty parameter list was made optional. The omission

LH: C# requires formal parameters?

DC: Who is asking for this?

BE: Community feedback, Domenic has mentioned it as well.

DD: Possibly due to CoffeeScript, but is nice to omit the parens where they aren't necessary.

BE: For this proposal, there is no issue in leaving the param list out

```js
let foo = => ...;

x = (y) // note missing semilicon
=> z
```

BE: but nobody does this.

DC: A concern, `x = y`, as written without the parens...

```js
x = y
=> z
```

(This is a valid Arrow Function: http://traceur-compiler.googlecode.com/git/demo/repl.html#let%20x%20%3D%20y%0A%20%20%20%20%3D%3E%20z)

AWB: No parameters: no LineTerminator here?

BE: If someone began with an arrow,

WH: I'm fine with this being an arrow function, but don't want this to depend on line breaking. I consider the following to be legitimate:

```js
  x = averylongid
    => foo
```

DH: This would introduce a new sigil that you'd have to worry about

BE: npm style is a thing. People actually put semicolons at the beginning of lines starting with + - / { [(, this would introduce => to that list.

```js
var i = "1"
;+i
```

WH: (laughter)

BE: you laugh, but people do this

WH: I thought I'd seen it all...

BE: Do we extend the short list of characters that can begin an ExpressionStatement

DH: It's not worth it

EA: I think it's worth it

BE: Writing "()" is too hard?

DD: It's ugly.

```js
it("should do something", => {
  // ...
});
```

vs.

```
it("should do something", () => {
    // ...
});
```

DH: I'd prefer not to have the hazard.

AR, DC: Agree with DH

Consensus/Resolution

- No consensus on removing the empty param list

4.3 Math.hypot

DH: The idea is that it returns the square root of the sum of its arguments, but we have 2 and 3 argument special casing.

http://wiki.ecmascript.org/doku.php?id=harmony:more_math_functions

It's an offense to mathematics if it's not: "The square root of the sum of the squares of all the arguments"

Defend the pristine purity of math. If it's only a 2-argument function, that's OK, but having it a 2- and 3-argument function is not good, just make it an n-argument function.

BE: Jason Orendorff outlined these problematic cases... (see es-discuss)

DH: this is no good man.

LH: Should be variadic

DC: Programming in general is an insult to mathematics. Because what we call functions are not what they call functions. One convention we have in JS is that calling a function with three parameters is the same as calling it with two plus undefined.

DH: but not for variadic functions. Since we want it for arity 2 and arity 3, cleanest way is to make it variadic. If you have 2 cases, yo have n cases.

Consensus/Resolution

- variadic
- call ToNumber on all actual arguments
- if one is NaN, no special behaviour, allow to fall out.

4.7 JSON.stringify and unpaired surrogates

https://mail.mozilla.org/pipermail/es-discuss/2013-September/033293.html

AWB: The concern is that JSON.stringify, as specified in ES5, if there is an unpaired surrugate

AVK: There's a bug in V8 where lone surrogates passed to the utf-8 encoder turn into CESU-8 byte sequences.

- What is the defined failure space

DC: The new spec is code points, so it could be unpaired surrogates

AVK: So it's 16-bit code units

DC: Yes
- JSON currently passes unpaired surrogates right through
- Suggest escaping them

AWB: Only for unpaired surrogates

DC: Will get through a UTF 8 Encoder, but will blow up

AVK: If you want the surrogates to make a round trip, then escape them

LH: Seems like it's saying, in practice, JSON will only admit into a JSON document UTF8

DC: (Question about range of affected)

AVK: only surrogates are affected

LH: were people running into this problem in practice?

AVK: Right now, roundtripping unpaired surrogates does not work. We could fix this by escaping them.

AWB: But this is an incompatible change from ES5, since the length of the string would change to include \uXXXX (or \uXXXX\uYYYY) instead of the surrogate itself.

AVK: (whiteboard)

```js
JSON.stringiyf("\uD800")
```

AVK: \uD800 is a single sixteen-bit code unit that is also a surrogate.

BE: this seems incompatible, so I say no

LH: this just seems like a single case...not sure why we'd patch just this case

AWB: this is about unicode/UTF-8 which is very common

LH: any time I serialize any JS string, I have this issue, JSON isn't necessarily what's in question

AVK: this is also about JSON as it's a network format

DC: why are people doing this anyway

AWB: e.g. because they're passing binary data as strings

BE: murphy was an optimist

WH: Note the CESU-8 spec, which is a mutant of UTF-8 that encodes individual surrogates individually (even if they're paired).

AR: if CESU-8 is bad, why isn't this fix bad?

AVK: you get subtle security bugs and it's incompatible with UTF-8. I don't really mind that we lose surrogates as I don't care about them

BZ: if you pass unparied surrogtates, you'll just get bytes. If people are trying to use the high bits to store binary data, there are going to be problems.

DC: the problem is that you're trying to pass 16-bit data through to UTF-8.

(binary data through UTF-8 requires separate escaping)

AVK: is a network format, it has strings

BE: JSON is more then a network format

?: If you want to pass binary data, just escape unpaired surrogates before serializing them as JSON.

WH: Won't work. JSON will escape the backslashes in the escape sequences and you'll get double backslashes.

Consensus/Resolution

- No Change. If you want to use lone surrogates you'll have to escape them after JSON.stringify().

5.1 splice, cross-realm Arrays and subclassing

http://esdiscuss.org/topic/array-prototype-slice-web-compat-issue

AWB: (whiteboard)

```js
let newA = a.slice();
```

AWB: it turns out people use this to make a copy of an array

In the general case, assume you have:

```js
class SA extends Array {}
```

let sa = new SA(10);

sa.slice(1, 5);

// What does this return?

// - a new SA?

// - a new Array?

```

It seems to make sense that when you slice a `SA`, you should get a new instance of `SA`

WH: (recalling Mark's points about Caja(?) use of slice) Array.prototype.slice.call(a)

AWB: This all makes sense, but the problem came up: what if "a" came from a different realm? Under ES5, slice() always gives you an array from the realm from which the slice function itself was defined.

EA: the expecation that you'll get an array is flawed since you are just calling "slice" on an unknown object. "slice" could return 42.

WH: Not what I asked. I was asking about the common existing idiom of calling Array.prototype.slice.call on an unknown object to coerce it into something known to be the built-in array. This breaks that.

```js

[].slice.call(a);

// This result will be an array from the [].slice.call realm

```

AWB: I've tweaked the spec to say:

- If the constructor I'm using to make the new instance is in fact a built-in Array constructor (not a subclass) from any realm, it then creates an instance of the Array constructor of the slice function.
- So it essentially does what ES5 does, but if it's a subclass it doesn't do what ES5 does, but ES5 does

RW: my concern was if NodeList was eventually to become a subclass of Array, what happens to existing code that does...

```js

[].slice.call(nodelist);
```

RW: it may solve itself? (the returned thing has the expected "shape")

(returned thing IS a realm-local Array instance)

AVK: we're not going to change NodeList (we're still trying to maybe put Array in its prototype chain), we're creating a new thing, Elements, which will work great.

RW: A new thing avoid breaking the old things.

BZ: well note that HTMLCollection can *never* get fixed in any way.

AR: just to clarify, if I have subclassInstance.slice(...), I get back what?

AWB: a subclass instance

(yay)

WH: If we adopt this, what would be the new blessed way to coerce something to an actual Array?

AWB: ES6 has *numerous* ways to force things to be an actual Array (not a subclass) from a given realm. Anyone who wants to force it can do that.

BE: yes, Array.from, or spread, i.e. `[...otherArray]`.

DC/LH: you're proposing a breaking change?

AWB: no, I'm actually removing a breaking change. For all existing cases, the behavior will be exactly the ES5 behavior.

WH: It is a change for the uses that expect Array.prototype.slice.call to coerce to an actual Array. They will need to change if they interact with any ES6 code.

(General concern to ensure the behaviour remains the same)

BZ: In the subclassing world, and I add method to Array.prototype, if I had an actual array, I will get an array from this realm.
```

BE: If the first arg to `[].slice.call(a)` (a) is an instance of a subclass of an Array in this realm, `a`s constructor will be used, otherwise, use

BZ: if the realm of the thisObj and the method do not match, create it in the realm of the method using the stardard Array constructor there.

AWB: Agree

WH: How do you find the realm of a Proxy?

DH: really don't want to go in the direction of relying to heavily on the realm

BE: What are you using to construct?

AWB: it's constructor property

AR: Mark counts on this behaviour to ensure that he'll always get "clean arrays" from something that may or may not be "dirty". He could accomplish this by using a frozen version of the method?

AR: Concerned about the Array "redirection", different prototype depending on what you call slice with

BZ: If you're doing "a.slice()", there is no problem, the result is a new instance of the thing you called slice

RW: Right, the issue arises when doing [].slice.call(arguments), nodelist, etc.

AWB: And these aren't arrays

(arguing the meaning of subclassing in ES5)

AWB: BZ's proposal of matching the realm of the function with the realm of the constructor is probably the right way.

DH: Seems like creating a patch for a scenario

DD: But this is what user code would do, using this.constructor

AWB: none of these are guaranteed, where you can wack the prototype...

<strike>

Consensus/Resolution

- When slice is call, what is the realm of the slice method and the realm of the constructor of the thisObj (since objects don't have realms, only functions do, we have to look up constructor).

- If they do not match, create it in the realm of the method using the stardard Array constructor there.

</strike>

DH: why not actually find out what realm an object was created in, instead of trying to infer it from this.constructor.

AWB: Is there any other way of creating an instance of this object without knowing what this.constructor is?

BE: In ES6, `class B extends A {}`

AWB: Need to know two things:

1. How to query what the constructor object is

2. What the protocol is for invoking that constructor

DH: So for slice, we have a default

AWB: Yes. Considering adding to collection constructors, a factory for creating

DD: Sounds like an unforgeable symbol that effectively is the same as .constructor, but can't be messed up like .constructor can.

DH: Object to something that is an approximation to what we want

BE: It's a reflective approximation

- Implementations do not want

AWB: What I am proposing and BZ:

What is the realm of the object I would create?

Is the object going to create an Array?

BZ: Seperate the concerns, two conversations about creation:

1. "How?"

2. "What?"

WH: Are you expecting to construct or call

(back to "how")

AWB: In my proposal you check if it's a constructor

BE: Opposed to that

DH: Have you audited these cases, how does it work for them all? Does it work for them all, except this case?

AWB: Yes. (The fix for splice is the fix for all of them)

BE: You've added a new reflection on the constructor

- Let's take this back and make it completely backward compatible

BE: I can write code that sets constructor, and now it works differently in ES5 and ES6.

AR: (mumbled something relevant)

BE: Don't think Mark would want the constructor check

AR: The dot operator is the root of all evil in JavaScript.

BE: I thought it was the NaNs...

AR: We've got a lot of evil.

DH: How much breakage are we talking about? Given the semantics Allen originally wanted?

AWB: under what situations does getting an array instance from a different realm break something? Well, if you've tweaked Array.prototype across your realms, that will make a difference.

BE: The odds are non-trivial that we'll break something

WH: If you're looking at the prototype of things, Caja will break. If you're looking at the constructor property, ES5 code will break for anything that has a custom constructor

AWB: Ignoring the Realm, unless we add some new operation on instances that reveal what Realm they are originally from.

BE/DH: need to go back and work on this.

Consensus/Resolution

- Needs revision of proposal.

5.2 Evaluation Order and The [[Invoke]] Operation

DH: Yehuda was interested in this.

5.3 @@unscopeable

AWB: Inheritance?

ARB: Wrong to apply this across inheritance

AWB: Changed in r18

ARB: How was this changed?

AWB: Own property

The remaining question, do we lookup on entry of with block or for every access?

WH: which one is slower?

AWB: every access

WH: OK, let's do the slower one.

(general laughter and applause)

ARB: (question about lookups that occur _up_ a prototype chain that might encounter an @@unscopeable)

DH: The semantics of with:
- There is an object that we'll do all lookups on

We're adding to that:
- Check the black list first

BZ: Consider you have two objects in your prototype chain, with two different @@unscopeable black lists

AWB: The first

ARB: What if you've put a "values" property directly on the instance?

RW: This was meant to fix that problem.

ARB/BZ: On the prototype, but what about an own property on the instance

(acknowledged)

ARB/DH: Discussion about predicted breakage.

BZ: Possibly as a descriptor?

RW: That was also considered

ARB: But if you [[Set]] it won't go away

```js
var a = [];
a.values = 1; <--
```

DH: (Recalling the issue that created this problem)

```js
function f(values) {
          ^-------------------------+
  with(values) {                    |
    values = 1;                     |
```

```
    ^--"values" here is pointing to-+
  }
}
```

But, if the parameter bound `values` object has a property called `values`, the `values` access INSIDE `with(){}` is accessed.

RW: The case that hits @@unscopeable in its current form is far on the edge. Ext.js is generally an outlier in their use of with(){} (in a world that's continuing to progress towards strict mode). Future versions of Ext.js won't have the offending with(){} code, so it won't break on @@unscopeable. Ext.js has also committed to evangelising and getting the patched code out to clients, it was just a matter of needing time to do so.

AWB: Someone could have created his own Array instance with names that @@unscopeable would hide. This would be also not be compatible.

BE: This is why I find find/fill kind of short, you might get collisions. ??

AVK/EA: But but but DOM

BE: Don't rush!

BE: Gecko did ship Array.prototype.values and backed out which is why we added @@unscopeable.

BZ: In the DOM we want Element.prototype.remove and friends which are problematic due to event handlers, which use with semantics.

BZ: You could have a proxy that does the same thing.

AVK: Ouch.

ARB: When an access occurs in with(){}, and a property is found, look at the @@unscopeable on THAT object

DH: Do we want to blacklist names or name-object pairs.

- Distinguish between allowing instance properties

BE: How does this change for MOP?

AWB: Changes from Get to GetOwn, etc.

BZ: DOM Proxies are nicely behaved.

...

AWB: Is anyone actually going to implement?

DH: This is easy for SpiderMonkey

ARB: This is easier then the previous.

BE: Concerns about changing the Identifier resolution for Proxy

AWB: Only for `with`

DH: If you want to not bypass the proxy mechanisms, you have have to give them a new trap to define how they behave when [[Get]] in `with(){}`

BZ: Have to give Proxy a way to know that it's happening in `with(){}`

(working through issues with document.forms)

hasOwn on @@unscopeable

DH:

- Walk the prototype chain
- foreach one,
- check if it hasOwn @@unscopeable
- hit a proxy, does the same

...This prohibits Proxy from participating in the object operation.

WH: Summarize?

BE: We have a problem adding names to mature the API. Let's move forward with @@unscopeable

AWB: For the @@unscopeable arrays I specify in the specification, should they be frozen?

DH: No, need to be able to polyfill feature additions.

AWB: You could always replace the array object.

BE: When in doubt, don't freeze. Why isn't it a Set?

DH: put down the freeze gun. Put it down.

EA: Set is the right answer, semantically.

AWB: Yeah.

DH: I don't really care.

AWB: You asked for a Set no, I think it should be a Set.

Consensus/Resolution

- Continueing with @@unscopeable, with changes w/r to lookup to be defined.
- Noted change:
  - When a property is found, look at the @@unscopeable on THAT object

- @@unscopeable
- a Set

5.5 Math.roundFloat32

AWB/DH: Offline discussion to rename to something more Math object familiar

Suggestions:

1. Math.fround()
2. Math.f32round()
3. Math.roundF32()

DC: This isn't a round operation

DH: It is, according to IEE 754

BE: fround. f means float, i means integer

AWB: What this effectively does, is "round" (observably)

Consensus/Resolution

- Math.fround

5.6 Backwards compatability, Unicode UTR31, and ES identifiers (U2e2f)

http://esdiscuss.org/topic/backwards-compatibility-and-u-2e2f-in-identifier-s

AWB: Traditionally, ES has said that Identifier are composed of unicode characters. There are changes to Unicode that rescind it's ability to be an Identifier.

AVK: According to Norbert, the character in question was not part of Unicode 3.0 and therefore not a change

(group testing identifier creation with U2e2f)

Consensus/Resolution

- No change
- SpiderMonkey: https://bugzilla.mozilla.org/show_bug.cgi?id=917436
- V8: ??

5.7 note in 11.6 WRT Unicode versions, update to Unicode 5.1

http://people.mozilla.org/~jorendorff/es6-draft.html#sec-11.6

AWB: (Norbert proposal)

Consensus/Resolution

- Remove: "NOTE 2 If maximal portability is a concern, programmers should only employ the identifier characters that were defined in Unicode 3.0."

5.8 line terminators in template strings. Should they be normalized?

(discussion re line terminators)

AVK: HTML parser does normalization, but e.g. DOM setters do not

BE: not normalizing would introduce more interop hazards than we want.

AWB: just to be clear, this is both cooked and raw form.

BE: yes. Add \r\n to the list of special things, alongside closing backtick and ${

- Propose: CR and CR+LF are canonicalized

AR: why isn't this just part of the data? Why canonicalize?

BE: it's not a byte string, it's a string.

BE: but raw should be raw

DD: no but then you have an interop hazard, because people will write code that works great on Unix, then some poor Windows user ends up with \r's in his strings that the Unix-using author did not anticipate.

BE: but you should be using cooked

DD: but a lot of the examples, e.g. on the wiki, use raw

BE: ah right, like the regex one, because you need all those slashes. Hmm.

BE: Python normalizes, Ruby normalizes

Consensus/Resolution

- Normalize CR, LF, and CRLF to LF

[WH: Is this consensus recorded correctly? I understood the consensus to be normalizing all lexical grammar LineTerminatorSequences to LF.]

EDIT

RW: Yes, and was re-stated for confirmation.

5.9 12.2.4 note says we decided (Jan 2012) tail calls only in strict mode. Is this still correct?

http://wiki.ecmascript.org/doku.php?id=harmony:proper_tail_calls

DH: function.arguments is poisoned in strict mode. This opened up doing tail calls.

LH: Are the tail call locations on the wiki still up to date?

DH, AWB: yes, to the best of our knowledge; let us know ASAP if there's something wrong with them.

- Discussion about tail call location opportunities

AWB: getting close to the point of getting tail call stuff into the spec (depending on how busy Dave keeps me); any comments appreciated sooner rather than later.

AWB: Implicit calls to accessor properties in tail positions, e.g. `return x.y`, and `x.y` turns out to be a getter, do we want this to be a tail call.

```js
return x.y;
```

If `x.y` turns out to be a getter, is that a tail call?

AWB: also consider `return x.y()` where `x` turns out to be a proxy; then it's not just a simple call, it goes through the proxy handler.

DH: More generally, if anything, syntactically not a call, turns out to be a call, do we include that as a tail call?

DH: basically, is the spec mandating something that makes too much work for implementations. The definition of tail position is not in question; it's the semantics of what things in tail position become tail calls.

LH: Function calls?

AWB/DH: and method calls

AWB: Calls in general

WH: new?

DH: no, `new` doesn't really work, because of the IsObject check afterward.

AWB: OK, what about the proxy case.

DH: I think it works fine; the semantics is just tail-calling the call trap.

STH: I agree with Dave; we're just making a tail call to some arbitrary code. The code we're worried about is in the call-*ing* function, not in the MOP operation/trap handler.

Consensus/Resolution

- Yes, the list is still correct.
- function.arguments in non-strict makes tail call effectively impossible.

(increasing insistence on break time)

AWB: but let's do point 5.11.

5.11 Disallow? let undefined; const undefined; class undefined {}; module undefined from "foo";

No one thinks that disallowing this is a good idea.

Consensus/Resolution

- Allow

BE: Let's make it a keyword inside generators! Just kidding.

4.2 Reconsider decision to make Typed Arrays non-extensible

http://esdiscuss.org/topic/non-extensibility-of-typed-arrays

AWB: Last f2f we decided to make typed arrays non extensible.

ARB: Chrome makes them extensible but I'm in favor of making them non extensible.

AWB: the most significant thing I saw is that if you create a subclass of a typed array, the reason for doing so might include adding some additional state, and so the only direct way to do that is by adding own properties to the instance of the subclass, and if typed array instances are created non-extensible, then subclasses couldn't do this.

DD: You cannot add indexed properties to typed array. It would be surprising if you could add identifier name properties.

DL: There are no objects in JavaScript that have this behavior right now.

OH: Certain restrictions exist for numeric properties. There is no other object that doesn't allow you to add an expando.

OH: Neutering is a performance issue as well. If we should make decision based on performance we should kill neutering.

BE: the real issue is not performance (although the thread may have misdirected in that direction).

AWB: we could go back to having numeric expandos, if you want...

BE: No no no no, there was a performance counterargument there, unlike for named expandos.

DL: (clarifying

```js
A = new ArrayType(uint8, 10);
s = new StructType({a: A, ...});
a = new A();
a.foo = 5;
s.a = a; // ??
b = s.a
```

DH: A "Typed Object" is effectively a pointer to a shared backing store. If those objects can carry addition state

OH: If

DL: With typed objects we can do

```js
Uint8Array = new ArrayType(uint8);
```

AWB: There may differences between TypedArray types

...

DH: There's an example, wherein two views on the same backing store will not be === or ==, but there are implementation strategies that could make it possible.

STH: It's easy to use getters where writing to one

I'm behind....

OH: Structs aren't expandable so therefore Typed Arrays shouldn't be expandable

BE: Do we need extensible array types

DL: All variable length array types to be extensible?

AWB: We'd have to have a form of class, whose instance properties are defined by a Typed Object

DH: Foresee a mechanism

WH: Prefix properties?

DH: Yes, if you have super type, its subtype is

OH: Index properties are seen first in ForIn

DH: Recalls discussion to allow index names to diverge, object model reform, etc.

DH: most beginner references establish that all properties are strings

RW: There is a lot of attention paid, in learning resources, to establish Array and Object as two different things, based on their property name "type". It's not until much later that realization is made that they are the not different, aside from special length property behaviour.

ARB: not allowing them is more conservative for now. We can always relax them later.

BE: that conservative argument seems strictly stronger than consistency arguments which always pick their preferred dimension of consistency.

WH: I haven't seem any good arguments for allowing expandos in arrays but not structs.

BE: the arguments are: it's useful; it works on arrays.

AWB: as currently specified, you can create a subclass of Uint32Array.

(Discussion of how buffer semantics work in subclasses. Decided it's not really relevant.)

AWB: the subclass wants to add new properties

DD: but it's OK to add prototype properties...

AWB: yes, but, most people represent per-instance state with own properties. Yes, you could use a weak map, that's the universal answer, but that's not what people do currently in JavaScript.

WH: composition over inheritance

AWB: but then you don't get any inherited behavior

AWB: Here's what I would advise. Use composition when you're composing base abstractions into a new abstraction. But if your new thing is a specialized kind of array, which may need some extra state, then that's not a composition type of situation, it's an inheritance type of situation.

DH/AWB: we could not make @@create create a non-extensible object, instead we would make the typed array constructor call `Object.preventExtensions` on `this`.

WH: extensibility breaks compositionality.

AR: this is a strange argument; you serialize and deserialize the object, losing expandos seems fine.

BE: serialization/deserialization is not the model.

DD&DH: so if you don't call `super`, or wait until the end of the constructor to call `super`, the constructor can extend with per-instance properties

OH: Have @@create produce an extensible object and have the constructor call `Object.preventExtensions`

DH: Yes

DH/AWB: What order does super get called?

DH: Seems like a solution, but not one that everyone is fully in support of.

DH: Points are not about speed of accessing properties of objects and more about layout of objects

BE: Most users of typed arrays are not naive. We should not be promoting these overal normal arrays.

AWB: Not even numeric?

BE: Define numeric. Unless you actually know what you're doing, you shouldn't be using these and we shouldn't be promoting them for these.

BE: Typed arrays are power tools.

(Float-containing normal array is almost equal in perf to float64 typed array in JavaScriptCore.)

OH: Agreed.

OH: Transfering large amounts of data to a worker.

??: Better not have any expandos.

BE: How do we make progress? Stand-off!

DH, WH: to be clear, it would be sad if typed objects and typed arrays were inconsistent. so if we say typed arrays should allow expandos, then so should typed objects.

DH: You can't put a variable-sized array in a struct.

OH: Do we have a ToArrayIndexedProperty somewhere? Might have been internal to JavaScriptCore.

AWB: The problem here is what array index meant when there was only the built-in array type doesn't fit with the typed array usage we have now.

BE: I don't see it.

Straw poll: 8 non-extensible, 8 mu (abstain), 1 (Ollie) extensible (I think it was moo, way more of a cow sound.) (https://en.wikipedia.org/wiki/Mu_%28negative%29)

OH: We don't see any reason to remove extensibility, you can always make them non-extensible

OH: The view of the JavaScriptCore team is that making them non-extensible removes what developers can do, whereas making them extensible does not have to cost and provides more freedom.

OH: examples of behaviour equivalent to TypeArrays with expandos:

```js
array = [1,2,3]
Object.defineProperty(array, "length", {writable:false})
array[3] = 0;
array[3] => undefined
```

Various DOM lists allow non integer expandos, but don't allow integer expandos

BE: The conservative argument is that if we're unsure we should make it non-extensible so we can make it extensible later.

AR: You can interpret it both ways.

DH: Non-extensible is conservative because you can change it later. It being extensible cannot.

DH: Extensible is probably my preference at this point, but we can't leave it unspecified.

BE: Due to duplicate bug of https://bugzilla.mozilla.org/show_bug.cgi?id=695438 -- namely https://bugzilla.mozilla.org/show_bug.cgi?id=828599 with its adding .stride to WebGL vertex typed arrays -- I am now in the extensible camp.

Straw poll:

Extensible: 10

Non-extensible: 6

STH: (points about the mental model of operating on the backing store)

DC: (points about the hazard of user code that abuses Array and creating a new hazard)

Consensus/Resolution

- Consensus deferred.

Sept 18 Meeting Notes

John Neumann (JN), Dave Herman (DH), Istvan Sebestyen (IS), Alex Russell (AR), Allen Wirfs-Brock (AWB), Erik Arvidsson (EA), Eric Ferraiuolo (EF), Doug Crockford (DC), Luke Hoban (LH), Anne van Kesteren (AVK), Brendan Eich (BE), Brian Terlson (BT), Rick Waldron (RW), Waldemar Horwat (WH), Rafael Weinstein (RWS), Boris Zbarsky (BZ), Domenic Denicola (DD), Tim Disney (TD), Niko Matsakis (NM), Jeff Morrison (JM), Sebastian Markbage (SM), Oliver Hunt (OH), Sam Tobin-Hochstadt (STH), Dmitry Lomov (DL), Andreas Rossberg (ARB), Matt Sweeney (MS), Reid Burke (RB), Philippe Le Hégaret (PLH), Simon Kaegi (SK), Paul Leathers (PL), Corey Frang (CF)

4.2 Reconsider decision to make Typed Arrays non-extensible (Cont)

RW: (recapping yesterday's discussion)

EA: Can we pick 3 people to champion a recommendation?

RW: Ideal

(support in the room)

Consensus/Resolution

- 2 People to come back with a recommendation:
- Dmitry Lomov
- Allen Wirfs-Brock

Post meeting update: Dmitry and I discussed the issue and our

recommendation is that Typed Array instances should come into existance being extensible. This is only about the objects created by the current set of built-in typed array constructors (or any subclasses derived

from them). It does not imply that fixed size array types introduced by the future typed objects proposal will necesarily also be extensible.

-- Allen

As part of that consensus, variable-length (but not fixed-length!) typed array instances that are part of a future Typed Object spec should also be extensible in the same way as current Typed Array objects. In that way, full compatibility and equivalence between say "Uint8Array" and "new ArrayType(uint8)" will be maintained. As part of typed objects proposal, we will also consider having a different "type constructor" names for variable- and fixed-length typed arrays (e.g. "new ArrayType(uint8)" vs. "new FixedArrayType(uint8, 10)").

-- Dmitry

4.4 Symbols

Dave Herman presenting (follow up slides)

DH: Symbols: object or primitive?

Open issues:

- privacy
- object or primitive

(1) Statelessness

- Symbols should not share state
- Encapsulates key and nothing else

(2) Cross-Frame Compatibility

```js
obj[iterator] = function*() {};
```

Another frame must also know what this `iterator` symbol is

EA: Workers?

AWB: Only an issue when you want to move a value...

EA: Case where you use a name as a brand (branding using public symbols do not work. Branding needs true private state.)

YK: can these be structure-cloned?

(3) Methods

DH: The one place that most objects can't have methods is prototypeless objects, but they can have instance methods. For most most of the interesting data (strings) there are things you can do on them:

```js

alert.call();

Math.sin(0);

document.getElementById("body");

```

DH: if you only allow things to work via functions with arguments, you are turning off a powerful tool

(4) Mutable Prototypes

Monkey-patching standard methods is a best practice

The evolution of the web depends on it

DH: This is important to the language

DH: mutable prototypes are the way that developers provide a consistent platform across user agents they don't control

STH: This is assuming that Symbol will grow methods

YK: It's actually an assumption that it _won't_

DH: If we freeze a prototype, we're closing the door to ever evolving the API and closing the door to user code experimenting

- No experience to show that freezing a prototype "works"

WH: By "mutable prototype", you mean add and change methods, not change the prototype

DH: Yes, changing the shape of the prototype object

AR: (general defense of mutability, in prototypes and otherwise)

AWB: Freezing the prototype can be undone safely, if there is reason in the future.

AR: this is not a conservative position, despite the claim.

DH: It doesn't matter, if we design depending on the invariance that the prototype is immutable then we can't change

(5) Non-Answers

(6) Shallow-Frozen Objects

```js
O.getPrototypeOf(iterator).foo = 12;
```

- Fails Desideratum #1: is stateful
- Fails Desideratum #3: distinct cross frame iterators

WH: What doesn't work?

DH: The standard iterator symbol would be different in different frames because they exist in different heaps

(6) Deep-Frozen Objects

```js
O.getPrototypeOf(iterator).foo = 12; // strict error
```

- Fails Desideratum #4: no evolution

YK: How does the current spec deal with the function.prototype linkage

AWB: The prototype is null

DH/YK/AWB: this is incoherent, doesn't work at all.

(7) Missed? Something about prototype-less wrappers, fill in from slides...

(8) JS already has an answer for this! Autowrapping of primitives

- typeof iterator === "symbol"
- Get/Call operations auto wrap
- Prototype state is global per-frame

"People think auto-wrapping is gross"
- provides uniform OO surface
- does so without runing immutability
- doesn't ruing API patchability
- need a solution for value types

(9) Remaining Issues

- [[ToPropertyKey]] of Symbol objects" auto-unwrap? Does it matter in practice?
- Worry about toString for symbols and Symbol objects? Does it matter in practice?

WH: We should be consistent with the way we already do things. We don't unwrap boolean wrappers when used in a condition; we shouldn't unwrap symbol wrappers when used to look up a property. If you use Boolean(false) in an if statement, it will evaluate true.

AWB: In ES5, there was special treatment of wrappers, e.g. in JSON.
- no reason you should have a wrapper value in most contexts
- I would say, don't use

YK: Is it important that === works cross frame or just the indexing
- Have a mechanism that allows them to

DH: A kind of object that overloads ===

YK: No, don't

- If you go with new Symbol() returns object with mutable prototype

DH: No "object" solution works because of methods

- Need to move to next slide

- w/r to toString, I can't construct a plausible scenario where this would be encountered

YK: Accidentally construct a wrapper

WH: People explicitly convert to string before doing an operation. It would be impractical to make symbols survive the various kinds of string conversions.

ARB: Lot's of existing code that has code paths that converts a value to a string to get property key, would subtly misbehave with implicit symbol-to-string conversion

(10) typeof Extensibility

We don't know that it wont break the web

MSIE "unknown type may simply be rare enough to be undiscovered.

Fallback: "object" with [[Get]] et al that behave like auto-wrapper? (Object.isValue()?)

WH: It won't break the web because existing scripts won't see the new type. Seeing one of these things requires changing the script.

WH: We said that array subclasses are new and safe because unmodified code won't see them break things like Array.prototype.slice.call; why isn't the same argument applicable to symbols?

AWB: when I did symbols originally, as primitives and not wrappers, it was a bug farm, because everywhere that assumed a primitive type, now had to explicitly account for the possibility of a symbol. The standard library in particular had to do this. In practice, anyone who is writing a general-purpose library would have to do the same thing. If you have wrappers, then the value "turns into an object" when you use them as such, which works fine.

ARB: I implemented symbols in V8 with wrappers and there were no places where they needed special handling.

AWB: yes I agree! Wrappers address most of the issues.

YK: (Concerns about existing code that might not be resilient to typeof)

DH: the most straightforward thing to do is extend typeof, but you can have code that's not resilient to new typeof values.

(More discussion of existing library code having to deal with symbols).

DH: Many new things in ES6 are objects with new APIs, so passing them in would violate basic interfaces expected. Whereas for typeof, many APIs will take any value whatsoever, and then figure out what to do via discriminating via typeof. So introducing a new typeof will break their assumption that they can handle any case.

JM: I don't think it's so different; they will fail in similar ways.

AWB: This is different; it's at the core of the language, and there is a different expectation.

DC: Yes, this is different. There is an idiom that will fail, and it's a common. `switch (typeof something)`.

DH: if the only conservative extension guarantee we can make is that if you don't use any new features, everything is fine...

DH: the difference is that there are APIs that say "I'll take any JavaScript value," they don't mean "I'll take any JavaScript value that was present in the ES5 era." This is different from saying "I'll take something with an array interface," and you passing in something like Map which happens to violate the array interface.

WH: you are picking the wrong strawman. We are introducing things that do keep to the array interface (array subclasses) but break existing idioms such as using Array.prototype.slice.call to coerce to an Array object.

DH: right, I'm picking Jeff's strawman, not yours.

AWB: the issue with slice() is about realms, not kinds of objects, or subclasses.

WH: The slice problem is not about realms. It arises merely if you introduce an array subclass and never use more than one realm.

WH/AWB: (arguing about what they're arguing about)

JM: You say it's clear when you pass in the wrong interface to an array-expecting vs. an all-expecting. Can you expand on why that's clear?

DH: Difference between an api says "I will take an arraylike thing" and I will operate correctly and API that accepts "any"

JM: but it's not about types for the "any" APIs; they usually just pass them through, e.g. a datastore API.

LH: Rare that any API says "I'll accept any"

DH: we're talking about parametricity, passing through the value vs. inspecting it. My experience is that I see type inspection of the type-any inputs a lot.

LH: How often does an API take type "any"

DH: A lot of JS programs don't protect against wrong values

WH: No such thing as an API reliably taking type "any" and portably doing anything useful with it. Suppose we later introduce a new Decimal primitive that obeys the IEEE standard in that a DecimalNaN is not === to itself. Would a map work with it? Code that exhaustively type-dispatches primitives simply must change once in a while.

YK: (recalling a point AWB made) people using typeof to defeat autowrap, tell the difference between a thing that auto-wraps and a thing that does not

DH: The reason we shouldn't be afraid, is typeof of serves the purpose ...

EA: If typeof primitive symbol returns "object" there is no way to distinguish a primitive symbol from a wrapped symbol.

DH: we'd have to add a gross check for distinguishing

- slipperly slope

WH: "the future is bigger then the past" and it seems we're trying to perpetually mortgage the future to fix a relatively transitory fear we're not even sure is real. The cost is forever having yet another mechanism to distinguish the types of primitives.

DH: Then there is the "browser game theory" argument, who will implement in the face of danger first?!

WH: (interrupts)

DH: I also like finishing my sentences. I'm willing to give it a try, but...

EA: FWIW, V8 is shipping typeof symbol under a flag and no bug reports

DH: under a flag is not the web

- Willing to take this back to SM implementors

AWB: Explicitly checking for "object"

DH: Existing code breaking, this needs to be considered.
- New code can find old code that can be fixed
-

RW: Will put Symbol through jQuery in test suite to see what "breaks"

Agreement that auto-wrapping is the way to go

WH: Yes, use auto-wrapping the way we know it

re: toString

DH: One way or another we'll have to have it, whether it throws or produces a string

DH: There are values that throw, like proto-less objects
- conversion to string is not infallable in JavaScript

ARB: implemented two toString methods
- Symbol.prototype.toString => throws to avoid the implicit coercion hazard
- Object.prototype.toString => applicable to Symbol, but have to be explicit

AWB: Plausible that Symbols could have a printable "name"

DH: Proposal summary:
- Symbols are primitive
- typeof is "symbol"
- standard Symbol prototype
- construct to create wrapped symbol
- Symbol wrapper object does not auto unwrap. ToPropertyKey will eventually call Symbol.prototype.toString which will throw.

ARB: for the record: that is exaclty what V8 implements

YH: to the implementers in the room, please make the error message when you use a wrapper very nice.

ARB: V8 gives explicit error message

DD: new Symbol() is a probable footgun; should it do anything at all? Maybe just throw? Because we don't have symbol literals, so unlike `new Number(5)` vs. `Number(5)` vs `5`, the choice is not obvious; people will try to do `new Symbol()` not realizing that this creates a wrapper.

RW: Agree.

ARB: but then it's weird that there is an object you can get access to, but whose constructor doesn't actually work.

DD: but the only way you can get access to these objects is via the `this` value inside `Symbol.prototype` methods, in sloppy mode.

DH: no, `Object(primitiveSymbol)` would give you back the wrapper.

DD: ah, damn.

AWB/ARB: Discussing valueOf returns, used in contexts where a numeric value is expected

DC: But string.valueOf produces strings

AWB: Do you anticipate future value types having auto-wrapping?

DH: BE has thoughts about typeof modifyability

RW: Defer until BE is present.

Agreement

AWB: How does user code define a "well known Symbol"? One that works across Realms?

DH: not sure

- Standard library, we can create something that exists everywhere
- Library code, not sure

ARB: Think this is a serious problem that needs to be addressed

DH: Proposal

- Agree that it's bad to have a Symbol with BOTH a private and public form
- Private by default? Public by access?
- Need to solve the remaining proxy leak problem

AWB: We've discussed and concluded that we're nowhere near a solution to the private state problem.

YK: Mark's solution inverts where the transaction occurs

DH: If private symbols really behave like WeakMaps and you invert where they live, then they are truly private

YK: You want the proxy to trap them.

Why?

AWB: It's just a property name.

DH: If you have access to the symbol

EA: What about iterator?

STH: If you want the proxy to have iterator behaviour, you need access to @@iterator

...

AWB: Mistake to conflate symbols, which are guaranteed unique property keys, with private state. There is a temptation to do it, but we run into problems. I want private state, but there are better ways to do it.

DH: I am pretty exhausted from years of this debate, but from all the things we have to decide, this has to be decided now. And I am willing to fall on either side of the fence (private vs. public vs. both), for the sake of resolving this, but we need to resolve it.

LH: agree; we can't leave this meeting without a decision on this. But Arv's proposal (GUIDs) does give us a path.

AWB: but Arv's proposal doesn't solve privacy at all; it's just about the representation of symbols (strings vs. real symbols).

RW/DH: strings as symbols have bad usability and problematic to use.

YK: No solution to the enumerability question (are symbols enumerable)

DH: yes, if we were going to go with this, we would just have iterator and create be a shitty string.

YK: worse, it would have to be a UUID published in the spec.

AWB/DD/AR: underscores are better than GUIDs.

DD/RW: Symbols not for private state, use a WeakMap. Symbols for uniqueness, move on.

DH: Arv, what is your issue with just having public symbols?

EA: they don't carry their own weight. They behave like strings, except for reflection.

(silence)

AWB: there is one difference. They are in a different meta-level of the system. There is no way that user data can coincidentally be confused for a symbol.

DD: also, symbols do not show up in `JSON.stringify`.

DH: This allows us to add a meta level. Just like `__proto__` in the past. Underscores are, until now, our magic feather that we wave around to say "This is the meta-level! This will not be confused with data!" And that's BS.

LH: unless you enforce the distinction, people will build abstractions that break through the layers.

EA: before we had `for-in`; then we gave people `getOwnPropertyNames` and people started using that; now we're going to give them `getOwnPropertyKeys` and they'll use that.

discussion about the string display

YK: debugger could recognize that it's a uuid and replace with a readable value

discussion about "__" names.

LH: if you use any english word, someone _could_ create a conflict. If you use "__", there is no accidental conflict.

YK: Using "iterator" or "__iterator__", no one can reliably ducktype for an ES6 iterator

STH: leaving for lunch, in favor of Symbols

EA: I'm still in favor of keeping symbols, sorry for derailing the discussion a bit; what helps is the possibility of removing `getOwnPropertyKeys`. That makes them secure in the absence of proxies.

LH: but then existing libraries cannot implement a real mixin that moves symbol-keyed properties over to a target object

(if symbols aren't given reflective capabilities)

(Brief discussion of making `Object.mixin` be the only way to do this.)

DH: `Object.mixin` that can transfer symbols plus proxies allows you to reimplement `Object.getOwnPropertyKeys`.

AWB: Still think we should have Symbols, getOwnPropertyKeys

WH: I object to getOwnPropertyKeys (because of too many historical layers: in every spec we seem to be adding yet another layer of enumerating the properties of an object with a we-really-mean-it-use-this-one-instead-of-the-previous-edition's vibe).

DD/AR/DH: (discussion of `Object.mixin` + proxies trick.)

YK: It sounds like we're slipping down the path of doing privacy with symbols again, and we're going to appease people for the wrong

LH: the concern for `getOwnPropertyKeys` is that people would just use it in place of `getOwnPropertyNames`. Maybe if we separate into `getOwnPropertySymbols` + `getOwnPropertyNames` that will make it sufficiently painful that people won't just use `getOwnPropertyKeys` together.

Consensus/Resolution

- Symbols are a new primitive type with regular wrapper objects
- typeof symbol === "symbol"
- implicit conversion to string throws
- new Symbol throws
- Symbols are public, not private ok that they leak to Proxy

- Symbols are unique
- Only exposed via Object.getOwnPropertySymbols instead of Object.getOwnPropertyKeys
- `Object.mixin` copies both symbol and string properties

Additionally:
- AWB commits to bringing a proposal for user defined well-known symbol registration

6. Post ES6 Spec Process

(Rafael Weinstein) - http://slid.es/rafaelweinstein/tc39-process

RWS: put together some thoughts after the last meeting with DL, EA, AWB, etc.

RWS: most of it is good except that it's date driven and the consensus requirements lead to a high-stakes game for getting features into the game.

RWS: The second problem is that with large-quanta spec releases, there's a varrying maturity level for proposals. Stable stuff is "held hostage" to newer features.

RWS: as we near a release, we end up with large pressure around things which may or may-not make it. Argue that this is destructive to feature quality.

RWS: we also have an informal process. It occurs to us that acceptance of features comes before details are sorted out. Implementers, therefore, lack a clear signal about when it's time to start implementing features. Might be unavoidable, but other groups show a different way (W3C, e.g.)

RWS: we also have a single spec writer who bears the full burden of authoring spec text.

RWS: a few ideas:
- decouple additions from dates
- put structure around stages of maturity
- what does each stage mean? Get clarity

RWS: non-goal: componentize the spec or break apart responsibility from the whole group. Also a non-goal to change the rate of evolution (necessarialy).

RWS: looked at how the W3C works and tried to extract the bits that seem to work well. A 4-stage process:

1.) proposal
2.) working draft
3.) candidate draft

4.) last call

RWS: At a (much more) regular interval, we'd post (smaller delta) drafts to ECMA.

AWB: do these stages line up with W3C terminology?

(sort of, not really)

RWS: proposals outline the problem, need, API, key algorithms, and identification of cross-cutting concerns. Also, and identified champion. Acceptance signifies the idea that the solution is something the committee wants to keep working on.

RWS: note that we don't explicitly slate a specific revision is targeted for a proposal. That comes later.

AWB: concerned that we might accumulate accepted proposals that there's no activity on. How can we structure a cull?

BE: as needed. FileSystem API as example.

RWS: the analog might be the "deliverables list" used by W3C -- removing something from the list on the wiki could be that thing

DH: Not to componentize? Seems like there is something of a componentization and that's the value?

RWS: don't want to abandon the goal of language coherence. CSS did this wrong and have lots of weirdness as a result. Non-communicating editors lead to pain. This model is different: everything merges into a single spec.

DH: how is this different to what we're doing now? Maybe this is a smaller tweak?

BE: What this does is adds more staging before "proposal"

RWS: this is saying the first stage doesn't have spec text, but the second stage does.

DH: Makes a lot of sense, might make sense to spell out the earlier "incubator" stage.

RWS: so there might be a stage-0, which is sort of the strawman we've had before

RWS: what we want to see at stage 2 is draft spec text. It can have early-quality notes, etc. but thought should be put into the text for the feature before we collectively accept the feature.

RWS: there are a couple of key things to look at: can we decouple spec editions from specific features? what are the substantives stages of maturity?

BE: quick question: the i18n spec was on a different track, is this only for core stuff

(quotes FakeAlexRussell??)

(sort of, might be a way to draw stuff into the main spec)

RWS: stage 3 is the "Candidate Draft". It signifies that the committee thinks the scope of the feature is set. We can incorporate changes, but the key thing is that implementations are potentially costly. This stage is a green-light for implementing and final feed-back

RWS: stage 4 is "last call draft". 2 implementations and an acceptance test that they pass. Once accepted at this stage, the draft can be scheduled for the next spec to be published.

RWS: what about dependencies? The committee isn't absolved of this. IT's up to us to manage them and there isn't any silver bullet. We need to make decisions.

RWS: thought a lot about linkage as a part of this. A champion's interests might work against the language (ducking dependencies, etc.). The committee still needs to advise and continue to look over the landscape.

(discussion)

AWB: implicit in this is redefining the role of the editor to be more of an EDITOR, and less of an author. Should probably have a role in advancing proposals.

RWS: so still a world where there's a single editor?

AWB: yes.

(general agreement)

PLH: Noting that some of the process order might be confusing/out of order, with regard to naming?

RWS: yes, "last call" means somethign different in W3C that doesn't map well

YK: the year might be a red-herring. The point isn't the date and the goal isn't to rush things under the wire.

RWS: (refers to Chrome release process) (not quite Chrome, but close and relevant: https://developers.google.com/v8/launchprocess)

AWS: some of the non-technical overhead can be offloaded

DL: part of the goal is to help offload the work, getting more people writing more spec text.

(notes that this happened for Proxies and O.o)

DL: inside the v8 team, we don't have a ton of visibility into the maturity of features.

BE: spidermonkey has shipped many things over the years, but at a cost

(discussion about implementations and style)

RWS: so we can imagine that you'd have different styles of implamentations at each stage? Makes sense.

(agreement)

AVK: the w3c is removing the last couple of these steps

PLH: there's a new draft on github somewhere

(some discussion that you need implementations, hence the new W3C process)

AR: the chrome process shows that some features might slip multiple features, and that's very good for overall quality.

AWB: are the criteria here entry or exit criteria?

(discussion)

WH: What about mutualy beneficial features?

AR: that's the dependency question, we talked about that

RWS: it's sort of arbitrary, but that exists no matter what. There's no silver bullet. It's the job of the committee to keep an eye on what's in flight. Not sure a process can ensure that we can do that well or poorly.

WH: not componentizing is good, but want to make sure that the process doesn't get in the way.

BE: true.

AWS: if we see things that are tightly linked, we might treat them that way

RWS: as I said earlier, the committee can choose to merge them

WH: is the intent that the spec will be written by many people? or a single author?

RWS: the hope is that we'll have more authors for sections of the text, and it'll continue to be the responsibility of the (single) editor to maintain quality.

YK: I've found it useful to go throuh the exercise of writing spec text

LH: I like that aspect of this proposal quite a lot

DH: I've found it useful to write things in pseudo-code when exploring many alternatives...there's a cost for writing it out that way

AR: things are meant to get more "specy" and improve in quality over time

(Reviewing previous approaches to specificying new features)

BE: I think ES7 should follow this

AWB: Yes

STH: As long as we're realistic about how much process change can really be helpful

DH: Smaller features can ship and large pieces can take the time that they need.

DH: need a way to post features in progress

WH: Difficult to do refactorings the spec if various folks write parts of it independently.

BE: Integration step left out? (eg. when does feature integration to the spec occur?)

- huge costs

- potentially huge conflicts

- need to identify necessary changes early as possible

WH: Concerned that the one-edition-per-year timeline is unrealistic both for us and for our users.

WH: Once-per-year would be too much of a moving target for users. For example, writing (and re-reading) books about ECMAScript would be difficult.

WH: Imagine trying to fast-track one edition per year through ISO, with yet another one done in ECMA by the time the previous one gets done in ISO. Also note that ISO has been known to generate interesting comments.

??: We don't need to send every edition to ISO.

??: Yes we do. They don't like it when you update an existing ISO standard and don't send them the update.

??: ISO likes their specs updated once every three years.

WH: How many simultaneous internal versions of the spec (the master Word document) would we maintain? Three?

AWB: One.

WH: Really? Let's say we'd plan to ship a new edition every December. When would we fork our internal spec to work on new features for the next edition while preparing to send the current edition to the General Assembly?

AWB: Every January

WH: Then we'd be editing two editions simultaneously almost all the time.

AWB: I can handle it.

WH: Yes, but can the reviewers of the spec handle it? We have enough trouble getting folks to re-read stuff as it is.

WH: Once every two years would be more reasonable.

Consensus/Resolution

-

5.10 Function parameter scoping and instantiation

Andreas Rossberg

[Slides](need to commit for a link)

Default Parameters/Arguments

Goals: Convenience Feature

- Readable!

Non-goal: subtle expressiveness

Should be able to understand the defaults without looking at the function body

ARB: Two Issues

- Scoping related to the function body

(examples of really weird cases)

Solution:

- Defaults should behave as if provided by wrapper function
-
-

Solution:
- Evaluate defaults in seperate scope
- Can see "this", "arguments" and function name (where applicable)
- Can see other parameters
- Cannot see bindings created in function body
- Cannot see bindings created in function body LATER (via eval)

Evaluation Order

```js
function f(x = y, y = 2) {}

function f(x = eval("y"), y = 2) {}

function f(x = (y = 3, 1), y = 2) {}
```

ARB: Preferably should be const bindings _in that scope_ (not the funciton body)

AWB: (describes the TDZ)

Solution:
- parameters have TDZ
- Initialized in sequence

WH: No distinction between a missing parameter and explicit undefined?

AWB: We agreed on that a long time ago.

BE: I thought there was agreement/discussion?

(referring to: https://github.com/rwaldron/tc39-notes/blob/master/es6/2012-11/nov-29.md#proposal-part-2)

(need slide examples)

DH: Most concerned with implicit rebinding

STH: The rebinding is only observable

(discussion re: mutation in parameter bound closures)

STH: Can fix this while preserving

ARB: Can change the "nutshell" to meet the needs of the concern items:

const => let

BE: In the example that binds

AWB:
- parameters are in separate scope contour
- visible to the body scope
- the body is disallowed from creating

- "namespace" for parameters

NM/RW: (agreeable points about curly brace boundaries reinforcing scope)

BE: Summary:

- Outer Scope
- Parameter Scope
- Function Body Scope

YK: (recalling names declared in parameter scopes being rebound in the function body)

AWB: I can express this with one Environment Record

ARB: Cannot, because of eval. A delayed eval in the parameter list must not see bindings from the body

```js

function g() {
    return 3 * 3;
}

function f(h = () => eval("g()")) {
function g() {
return 2 * 3;
}
h();
}

```

AWB: Agreed

DH: (post-clarification)
- Two Scopes
- The Function Head/parameter list
- The Function Body

In the function head/parameter list, cannot see the scope to the right (the function body).

AWB: Any new syntax in the parameters, changes the handling?

(vast disagreement)

AWB: The spec currently says var-like bindings. If you have new syntax, they're still var-like

- Duplicates are illegal
- Rules about redeclaration
-

```js
// If...

function f(x) {
var x;
}

// changes to...

function f(x = {}) {
var x;
}

// No difference.

// But changes to...
```

```
function f(x = {}) {
let x;
}

// Error for redeclaration.
```

(clarification re: nothing changes var bindings to let bindings)

WH: (whiteboard) What is the value of y in this example? 5 or 2?

```js
function f( x=(y=3, 2), y ) {
    console.log( x, y );
}
f(undefined, 5);
```

(discussion without a clear resolution)

CF: What about:
```js
var y = 2;

function f( x=y, y=3 ) {
    console.log( x, y );
}
f();
```

BE & Others: `y` is shadowed result is (undefined, 3)

WH: What is the value of y in this example? 2, undefined, or 5?

```js
function f(x = (y = undefined, 7), y = 5) { ... }
f(undefined, 2);
```

AWB: The original value of the parameter is used to decide whether to default it or not.

BE: Surprised. Unhappy with having to store the original values of the parameters, thereby making for two copies of each one.

AWB: Already need to do this for the arguments object.

BE: The arguments object is easy to statically detect. These are more insidious.

(no clear resolution)

ARB: Fundamendally these are mutually recursive bindings.

BE: We agreed on two scopes. Head and body.

- If another parameter has a default?

Consensus/Resolution

- Two Scopes
- Head/Parameter List
- Body
- Temporal dead zone?
- Details unresolved?

4.5 Modules Update

Dave Herman

[Slides](need to commit for a link)

Generic Bundling Slide

(Debate about hash as the delimiter. Agreement that this discussion can take place elsewhere.)

DH: the browser loader is not something that belongs in Ecma-262. It's a separate spec. We can do it concurrently. We definitely want to start now and get feedback early, but it doesn't need to block ES6.

(Discussion of confusion on parsing vs. evaluation timing. Custom loaders can implement the desired esoteric use case; see caching slides.)

DH/LH/JM: Use case under discussion is lazy module execution, like AMD bundles or previous named module declarations. If you have a `console.log` inside a module, is there a way for that not to get executed?

DH: we may need to check to ensure that is possible, but it probably is. And the simplification of removing named module declarations still seems worth it.

4.6 Unbound variable checking

Dave Herman

DH: Proposes that if m is an imported module, then m.bar should be a compile-time error if the module doesn't have a property named bar.

WH on whiteboard: Should this be a static error in that case?

```js
module m from "foo";

with (a) {
    m.bar;
}
```

?: Modules are in strict mode and don't allow 'with'.

WH: But this isn't a module; it's just referencing one.

Module loading

DH: it used to be that `<script async>` would be able to do I/O, including `import` declarations; I've relaxed that. Now `<script>` can do that.

DD/DH: (clarification that you can use module syntax in scripts, not just modules>

BE/DH: (discussion of allowing `<script>` without `async` to load modules.>

AR: note that inline scripts with `async` or `defer` attributes *currently* do not impact execution or parsing. This may change in the future.

JM: if people want to use `import` in a synchronous script definition, that should be OK; just throw

DH: that was the direction I was moving, but LH was objecting to. And DD has an interesting point that if we don't let `import` happen at the top level, that would work well too.

STH: What do we like?

YK: Adding a form to HTML that says "this is a module." This reduces the need to allow `import`s in scripts.

BE: that would mean we're betting on getting something into HTML

DH: yes, but you could just use the loader API.

BE (whiteboard): four cases

```html
<script>(1)</script>

<script src="...">(2)</script>

<script async>(3)</script>

<script src="..." async>(4)</script>
```

WH: how do you load a module without async scripts?

YK/LH: `System.load("module")`

WH: and you wouldn't need to import `System` or similar

DH: no, that's just a global

WH: but we have features that require modules, e.g. `iterator`

DH/YK: yes, but you can just do `System.get("std:iterator").iterator`.

WH/DH: if it's hard to use a module inline in the page, then it's hard to write the good code we want them to write.

DH: this is something that needs to happen for multiple reasons, so it should happen in HTML.

YK: `import` in top-level scripts doesn't give us modules in top-level scripts, only `import` in top-level scripts.

JM: so how do you enter the module system from HTML?

DH: two ways. The loader API, or the hypothetical `<module>`.

BE (whiteboard); top level script looks like

```js
let { keys, values } = System.get("@iter");
```

DH: BTW JS practictioners, I'd like to reiterate if you have concerns about the standard module system.

LH: Implementers will ship iterators before modules, so we need a way to get at these things more easily.

DD (jokingly): We can just use a proxy to trap `@@iterator` in the meantime.

DH: I really think this how-to-enter-the-system conversation can occur outside TC39.

BE: so we can provide two top-level environments.

BE: OK, this is all about separation of standards-body concerns.

DH: and this helps not block TC39.

(Discussion somehow turns back to `<script>` vs. `<script async>` getting module-loading abilities.)

BE (to LH): so you're worried about an attractive nuisance, people doing more synchronous loading than they should

LH: Well today, `import` always succeeds, but with this proposal, it's order dependent, like today's `System.get`.

WH: <module> as a new HTML element won't work due to HTML parsing issues. Note that scripts contain un-HTML-escaped <'s (and even larger chunks of HTML tags) while other HTML elements don't. An HTML

parser wouldn't know how to properly skip past an element (such as the proposed <module>) that it doesn't know about.

DH: I think `<script type="module">` or similar is going to be necessary, for many reasons.

DH: so to recap, there's the two possibilities: allow gradual integration via `import` etc. in scripts, or the green path where you enter the module system once and then are there.

JM: Facebook wants both, so we can do initial page load and async load.

DH: that's fine, you can do that with `System.set` in the initial page load.

DH/BE: (Agreement that this should go in other standards bodies.)

Back to Static Checking

LH: back to static checking?

BE: you have to do label checking. It's not that bad.

ARB/BE/DH: we have to implement to find out.

BE: how much parsing do you have to do?

ARB: so that's in the pre-parser for V8

BE/DH/ARB: (discussion of V8's pre-parser)

DH: somewhere in between a reader (along the lines of SweetJS) and a parser, and that's what I don't understand.

ARB: it's a parser, but it just glosses over a lot of the grammar.

ARB: to be completely honest, we would like to get rid of this thing.

BE: so we won't know if adding this static checking for modules has implementation consequences, until implementers actually go implement it. So if they have appetite for it, we should try to do that.

DH: JSHint or TypeScript could do all these things... We need to at the very least provide the basic foundation. But that would shut the door on further static things.

BE: V8, do you guys have an appetite for trying it?

ARB: I'd like to try, but not sure if it's possible within the ES6 time frame.

BE: and what about Chakra?

LH: we can try it, but we don't know...

DH: it would be OK with me to close the door on static things like guards.

BE (to ARB): wait I'm confused. If you're doing import/export checking, aren't you doing about the same work you'd be doing for full static variable checking?

ARB/LH: no

DH: import/export is top-level only; you don't have to walk the full AST

LH: you would have to freeze the global environment at the point in which the static checking happens, and test against that

DH: yes, that's right

BE: OK, so maybe it's enough to have import/export checking. That spot-in-time check could be a problem. Yes, this is a problem for monkey-patching.

DH: every time we go through these cases it takes hours to remember the global object semantics.

AWB: I thought we concluded a long time ago that we had to preserve global semantics.

DH: clarifies: only talking about within the body of a module.

- Check the script against the current state of the Global object at compile time
- This is an unsound and incomplete analysis, but, it's one that you can program to.

BE: so if we say that module bodies do not have this type of static name checking, we're closing the door to guards, hygenic macros, type checking, ...

WH: how does it close the door to guards?

DH: we always talk about guards as if we knew what their semantics were...

BE: OK, well, how about truly static stuff like types or macros.

DH: my experience in ES4 was that it was fighting with the dynamic aspect of the language

WH: in Lisp we have a multi-level time-of-execution (i.e. eval-when) system... it was very messy...

BE: I think static types and static metaprogramming as an option are shown to be not possible, really, via the fact that TypeScript and Dart are both basically WarnScript.

DH: I think that it's been shown that tooling is generally how the web solves this problem.

LH: and we could do this outside the language itself, the opt-in could be e.g. opening the debug tools instead of being in a module body.

STH: But, nobody's said that this is a horrible feature, there's just some implementer reluctance.

DH: JSHint works fine; modules alone will allow JSHint's undefined variable checking to work without having to provide a large list of globals.

LH: we've started creeping a little bit toward doing more static analysis, but this would be a big step.

DH: what do you mean static analysis.

LH: I mean more early errors. We added more in ES5, e.g. duplicate variables. ES6 has added more with `let` and `const`. This is the next big jump. It's not clear where that's trying to go... We could go much further, we could build the whole linter into that point.

DH: I have years of experience writing Racket code, which works exactly like this. Once you're in module code, you have static variable checking.

LH: but no global object in the scope chain.

DH: actually kind of, but yes, people don't use it nearly as much as on the web.

DH: The static variable checking is both unsound and incomplete; the former is because of snapshot-in-time globals, and the latter is because of the halting problem.

WH: I want a way to get static variable checking but also monkey patching. Perhaps declare which global bindings you might want to monkey-patch.

checks on import/export

Sept 19 Meeting Notes

John Neumann (JN), Dave Herman (DH), Istvan Sebestyen (IS), Alex Russell (AR), Allen Wirfs-Brock (AWB), Erik Arvidsson (EA), Eric Ferraiuolo (EF), Doug Crockford (DC), Luke Hoban (LH), Anne van Kesteren (AVK), Brian Terlson (BT), Rick Waldron (RW), Waldemar Horwat (WH), Rafael Weinstein (RWS), Boris Zbarsky (BZ), Domenic Denicola (DD), Tim Disney (TD), Niko Matsakis (NM), Jeff Morrison (JM), Sebastian Markbage (SM), Oliver Hunt (OH), Sam Tobin-Hochstadt (STH), Dmitry Lomov (DL), Andreas Rossberg (ARB), Matt Sweeney (MS), Reid Burke (RB), Philippe Le Hégaret (PLH), Simon Kaegi (SK), Paul Leathers (PL), Corey Frang (CF), Mark Miller (MM)

(discussion re: destructuring concerns)

AWB: If you want to not throw for no value you have to define the default value.

DH: That's not YK's position. He wants to not throw.

DH/AR: The way you pull things out of an object is to do a [[Get]] wich does not throw and returns `undefined`

WH: What about `{a:b, c:{x}} = {}`? If you emulate the [[Get]] model, you'll still throw on a two-level destructuring pattern. Not clear what the useful point of sometimes soft-failing and sometimes hard-failing, even within the same pattern, is.

DH: That is not the same issue.

DH: In the case where the thing you are destructuring is an object, and the property you're looking for is not there, it should not throw. That matches existing JavaScript

```js

{ a: { b: c } } = {};

```

The inner object is the source of the error.

ARB: Confused because notes different from what just said.

AWB: Currently spec throws unless a default value is provided.

YK/DH: Not happy with that outcome.

RW: (recalling agreement between AC/YK/RW at last meeting on feeling that we're not following expected JS behaviour)

DH: The obvious case is using an object.

- What does the syntax most naturally correspond to?

- Looking for smooth refactoring paths

ARB: I don't buy that it will be common to refactor like this

ARB: Common bug that you get `undefined` for `o.x`

DH: That is just how JS works and we cannot redo JS.

AWB: I can represent YK's position. Personally fine either way. But we need to decide. We cannot keep putting this off.

DH: I believe we would dissappoint the community if it threw. It is just too different from what they are used to.

AVK: My recolection was that we woudl go with no exception and maybe add a `!` in the future.

LH: Ultimate consensus at last meeting was fail soft, waiting for ARB to object.

ARB: Other consistency arg with a future formal pattern matching

AWB: Yes, but for pattern matching we'd have something else

LH: If pattern matching used something else, and you were in that context, it's not a stretch to tell people there are new rules in that context.

ARB: Results in two semantics for one syntactic class (patterns). Bad for consistency and language economy

DH: Whatever familiarity from other languages and contexts, we need to align with JS and align with fail soft

AR: (to ARB) the practioners in the room are consistently disagreeing with your position.

RW: In a pattern matching context it is fine to do things more strict. People will not be surprised by the difference between destructuring and pattern matching

DD: (recalling recent extensive destructuring experience)

DH: Opposed to having two different semantics. Throw in destructuring but fail soft in [[Get]].

JB: What was the problem with `?`

DH: Default behavior is backwards.

AWB: There plenty of unresolved syntactic and semantic issues and not enuogh time to get them done in ES6.

JB: and `!`?

AWB: No bang for ES6

RW: opposed to re-appropriation of `!`...

ARB: I think this makes for a worse and more error-prone language. But acknowledge that I am alone and I will not stand in the way of this.

Consensus/Resolution

- Throw if not an object
- Then do a (fail-soft) [[Get]].

9. Promises

Domenic Denicola

https://github.com/domenic/promises-unwrapping

DD: Consensus on AP3. Some issues with extending toward the future. Some bugs in the DOM spec. Tried to fix those.

MM: Recommending that TC39 adopt promises-unwrapping so that w3c can proceed, and TC39 also get consensus on adding `.done`, `.flatMap`, and `.of`.

AVK: promises-unwrapping is wanted for shipping in browsers. A lot of specs that rely on promises and we'd like a blessing.

- AP3 was initial consensus

- changes were made to make new consensus

MM: Can we agree on promises-unwrapping to move forward?

STH: The promises-unwrapping spec, in that it doesn't include ... (Google Hangouts misbehaves.)

MM: (explaining semantics and benefits of flatMap etc)

DD: without flatMap they will unwrap on the way out

STH: unwrapping?

DD: input side doesn't unwrap, only the output side

STH: Then I'm happy with this.

MM: No dissent from promises-unwrapping with the addition of .done, .flatMap, .of

WH: What is the unwrapping doing

DD/MM: explains that unwrapping occurs as long as there is a then() on down until there is no then()

ARB: (to Sam) I share the compositionality concerns. Are we sure there is compositional abstraction if you use two levels of abstraction?

STH: if you ever write .then, your system is not going to be compositional where promises are a data type (or you'll have to do extra work)

WH: What is a then-able promise?

DD/AVK: Just an object with a then function and you assimilate. It's "promise like".

WH: .then does what?

DD: .then is how you extract values

MM: (explains unwrapping again)

WH: What is .flatMap?

MM: A promise accepting another object, causes the .flatMap to

CF: it's "then" without magic

MM: it's lower level, .then is built on top of .flatMap

ARB: .flatMap is parametric and does no magic on it's values, where .then does

LH: the only way you can convert a thenable to a promise is return it from a promise. `Promise.cast` and `Promise.resolve` will not convert a thenable?

MM/DD: no, `Promise.cast` and `Promise.resolve` work the same way, storing any thenables as their value, and then the unwrapping happens when you call `.then` on the promise who has that stored as its value.

WH: What is ["The ThenableCoercions Weak Map"](https://github.com/domenic/promises-unwrapping#the-thenablecoercions-weak-map)

MM/DD: (explanations of security concerns)

- No code contributed by the arbitrary object will execute during that call

- assimilation of thenables was constructed so that the object cannot cause side effects during the operation

DD: It's clearer when the code intends to run async, vs. some code running when assimilation occurs and some code later.

MM: Then I should talk about .done now

AVK: I think we have consensus on promises-unwrapping, and can defer `.done`.

MM: declaring consensus now is crucial to unblock the DOM. If we can defer `.done` I am fine with that.

(General agreement that promises-unwrapping with .flatMap and .of has consensus and .done can be deferred.)

LH: Will need to add cancellation capabilities

- Want to make sure that if we're sticking this in DOM apis, make sure there is back-compat safe to add them

AVK: I believe that Mark and Domenic have given plenty of thought here

DD: (confirms)

MM: Notes that test262 will need to be extended to support async testing

DH: This is really well developed and thoroughly spec'ed... what is the possibility of getting this into ES6?

(murmurs of insanity)

AWB: We're close approximate spec deltas here. Not quite cut and paste, but encourage that we might be able to fill in the editorial aspects.

- What about the event loop interaction

MM: I think the right precedent is Object.observe, it was very well written, very complete and we adopted to ES7 (for as much as that means)

DH: Doesn't need to be tied to the event loop

- event loop is very clear.

- would love to recast the loader api in terms of promises.

MM: That's a better pay off

AR: Not quite that simple... in many cases result in void return types

AR/DH: agree that this is better overall

AWB: More confident about Promises, vs other features. If editorially practical, we should try.

AR/AWB: No syntax, so no issue there.

PLH: Makes life easier for w3c specs as well.

AWB: What about an ECMA technical report in the interim? Or an independant spec in the interim?

DH: In practical terms, that would mean I couldn't use them in the Loader api?

MM: w/r to synergy between module Loader and Promises

- how much of a difference does it make, if you could rephrase the api in terms of promises?

STH: Many methods would change to use promises, a few cases would be drastically simpler, and all cases would be improvements in useability.

MM: I would be over-joyed to have this in ES6

DH: Most important to this: Domenic, Anne and whoever need to provide Allen with complete works as needed.

WH: It's weird that if having a "then" property that's not a function is equivalent to not having a "then" property (the object is considered non-thenable), but having a "then" property that throws prevents the object from being returned from a promise. Too ad-hoc.

WH: Let's say we introduce structs where if you mis-define fields it throws?

DD: if you introduce changes like that, you'll have to re-factor checks throughout the spec and .then can be refactored in kind

DC: To be clear, a thenable is:

DD: An object that has a .then property whose value is a function (is callable)

WH: understood (but don't like it)

DC: ok

AWB: A bit of legacy around "callable"...

(Discussion about detecting then properties)

DD/MM: JSON.stringify precedent: determining whether to return a property based on whether it's callable

AWB: JSON.stringify is filtering...

DD: But same meaning

AWB: These callability tests are unnecessary?

DD: Proven to be necessary

WH: To avoid objects with .then that isn't callable... Why aren't we using [a well known symbol] instead of the string "then"?

AR/DH: There is no way we can introduce this feature that has a change like that.

- A lot of existing code to interop with

DC: "then" is the wrong word.

MM: for a long time I fought for "when", but there is too much web-reality that calls it "then" and it wasn't worth fighting

WH: What happens if a thenable doesn't call onFulfilled or onRejected?

DD: then it stays forever pending. This is a valid use case, e.g. a server that never responds to a request.

DH: and it's actually a really nice zero of the promise algebra!

Consensus/Resolution

- fast forwarding Promises into ES6 as per https://github.com/domenic/promises-unwrapping
- Postpone with option of revisiting
- cancellation mechanism
- discussion of done method

7. Object.observe status report.

Rafael Weinstein

[Slides](http://slid.es/rafaelweinstein/object-observe-sept2013) <--- etherpad fucks this up :(

Discussion related to how nested observers should chain.

AWB: Maybe have `performChange` do take one more parameter, that is the record a function that calls notify.

NM: Or have `performChange` return the record.

AVK: You can skip the `object` in the record because the notifier knows which object it is working with.

MM: Does not seem like a good path to not handle expecptions???

RWS: The mutation records from array methods are about the intent to mutate the object. It cannot tell what the new state is of the object.

MM: If somethings fails, and you try to perform the same operation on a replica you will get the same failure on the replica.

RWS: I attempted to do the work and this what I intended to do.

MM: I'm fine with this as long as it maintains the ability to keep a replica consistent.

AWB: Would it be ok to not record property changes on array propert changes.

WH: What kind of a change record would "sort" generate? In particular, how would the change record describe how the array was sorted (ascending, descending, by what key?)?

RWS: If the array only said it was sorted then the code would need to keep a copy around to know what happened.

WH: In that case would reverse also emit a sorted change record?

AVK: Is "sort" proposed.

EA: No

(discussion about observing changes of attributes such as making an object non-extensable)

AWB: It is uncommon to care about property changes for lists.

AVK: <input type=file>.files might want to use Array.observe. It only cares about the items in the array.

AWB: It seems strange to use observer for this use case.

DD: generally DOM has a lot of things where the only difference from normal ES constructs is that when the object changes, you need to update something on the user's screen. New subclasses seems unnecessary, there's no new API.

AVK: Considering using array or a small sub class. Reusing array as is easier because you get a lot of things for free.

AWB: Use more specific class than array.

(Discussion about Array.observe vs. Object.observe.)

RWS: Allen, I think what you're saying makes sense, and it's a specific instance of a more general thing of filtering, which we may want for performance. Let's defer that.

CF: An API question---what about { new: newCallback, updated: updateCallback, ... }, instead of (callback, ['new', 'updated', ...]).

DH: yes, callback-last is definitely important

RW: (explains in depth the benefits of this)

RWS: I'm not especially excited about separate callbacks, because often you want a stream of change records, and not to react individually to each of the operations.

RW clarifies with some code Corey's proposal:

```js
// either

Object.observe(foo, {
  updated: function() {},
  deleted: function() {}
});

// or

Object.observe(foo, function() {});
```

```

RWS: This is an antipattern. We don't want to split the callback like that because the change log is the important part and if you split it it is hard to get ordering right.

RW: The misunderstanding: the list of change types is a "white list" of change types to include in the change list, not a 1-to-1 "events to handle" list.

WH: Want the names to be consistently present tense: new, update, delete, prototype, reconfigure

RWS: prototype is used when [[Prototype]] is changed

WH: how often do you observe an object whose prototype chain changes?

RWS: well, a common use case is using the prototype chain to represent concentric scopes, e.g. Angular

RW: It is valid to want to observe changes in the prototype chain, but I don't think Angular is a good supporting argument.

DD/RW: Object.setPrototypeOf is the supporting case for observing prototype replacement

DD: so then why not include observing the changing of extensibility

AR/AWB/RW: I think we need that for completeness anyway.

MM: yes, any mutable state should be observable; as long as it is observable by polling, `Object.observe` should work.

RW: agreed, you could definitely implement it.

Moved on to "Thought Experimental" slide.

WH/DD: the names on this slide are weird. "deleted" doesn't work (it's already used by normal objects). "set" vs. "updated". It seems like namespacing is necessary.

RWS: Agreed, there is a namespacing issue.

WH: Would prefer to keep the simple notification names ("splice", "set", etc.) to match method names that generate those notifications. It would be bad if we got into a pattern where method Foo generated ArrayFoo notifications when used on arrays, MapFoo notifications when used on maps, etc. This would be an annoying
```

abstraction leak for observers who don't care which particular data structure is used to store the things being observed.

RW: Map and Set operations have potential to be deceptive; since the actual data is held internally, freeze operations have no effect (freeze is on the surface for tamper proofing), so there might be a situation where a Map or Set is "accidentally" assumed to be locked down but is still observable. FWIW, I do like the addition of change observation for Map and Set.

MM: Freeze is not about freezing the object, it is about making it tamper proof. I think we can postpone this to ES7.

RW: (to RWS) we can talk about this more offline

AVK: it would be nice if there was a recommendation for how to do namespacing, for other specs etc.

WH: Asks about ordering semantics

RWS: there is an unresolved issue about ordering of different types of work in microtasks (promises vs. `MutationObserver`s vs. `Object.observe`); this is still undecided.

... Moved on to the performance slides.

WH: The slides are comparing the proposed language mechanism to polling, which is a bad choice for the comparison control group. If I were implementing observers in existing ES5, I definitely would not do polling; I'd set dirty flags and keep a list of dirty things. That should be the control group for the performance comparisons.

RWS: the point of these graphs was not to show anything particularly interesting, but to show that there were no major surprises awaiting implementations.

AR: What do you need from this group? How close are we to being "done"?

RWS: Got good feedback on a few things to change. Maybe next meeting we'll have something that's really "done" and we can't go any further without implementations.

AVK/RW: just be sure to update us on es-discuss when you make changes.

Consensus/Resolution

- failure cases, what to do when an exception happens midway through performChange
- change type naming, eg. "sorted" => "sort"
- 2nd/3rd argument order (offline discussion)

- Post to es-discuss when wiki page is updated.

Licensing Concerns?

AWB: there was some discussion on the mailing list...

RW: I am pretty sure that was a troll.

STH: So this same guy actually came on the scheme mailing list and behaved similarly. It seems he just wanted to upload the PDFs somewhere, and did not actually care about the contents of the spec.

8. Data Parallelism

Dave Herman & Niko Matsakis

DH: I wanted to explain why the issue of sequential fallback is not as simple as "we should just throw" for synchronous code.

NM: I want to separate out throwing on non-parallelized execution vs. non-parallelizable execution. The former is not what people want.

(General agreement.)

DH: yes, you want the engine to be able to make dynamic decisions about whether parallelism is profitable.

NM: it turns out there are many reasons why an execution may be non-parallelizable, not all of which a user should concern themselves with. There are implementation constraints that make it very hard to parallelize in some cases, but in theory they should be parallelizable. For example, in SpiderMonkey, string operations: they are currently implemented in a very scary imperative way that is hard to make threadsafe, but from a high-level perspective it should be obviously parallelizable. (It's not mutating shared state.)

NM: Our conclusion was, we would instrument our JIT compiler to generate parallel-safe code, which needs a warmup; we'll run sequentially for a while, before we're ready to *try* parallel execution. It's going to be hard to implement a parellization strategy that doesn't work like this.

ARB: I totally agree.

DH: Cannot get to paralelizing until have done some serial exucution to gather information.

NM: Implementations will grow the set of code it can parallelize over time. Cannot force any constraints on the closure. The alternative would be to instrument the entire the engine ot keep track of what cannot be parallelized.

NM: The other option is to formalize what can be parelized in the spec. For the end user they will still not know what will run in parallel.

EA: What is preventing engines for running Array map/filter etc in paralel if it cannot be detected.

DH: It would be an interesting thing to try out.

NM: There are other methods like reduce that cannot be parallelized. We therefore still need the par* methods.

ARB: Throwing if it is not parallelized is not sensible because it is too hard to specify what things can be parallelized.

DH: The order is non deterministic. That is the big difference. And this makes it easier to parallelized.

NM: THe order is only crucial for reduce.

WH: Even if an operation has no side effects, if several of the constituents of a map throws, you might get the wrong exception.

WH/NM: (discussion about definition of side effect, whether throw is a side effect.)

NM: 1) mutation of external state; throws; and straying into native code (that we don't have a safe version of).

LH: if I'm interpreting this correctly, it sounds like the whole parallel JS thing becomes less of a standards thing, and more of an implementation concern, except there will also be some non-deterministic parallelizable array method.

DH: but it's also important to give the parallelism recommendations teeth because then authors can depend on it.

NM: it's also useful to have a specification available for users to read to understand what works where. It doesn't effect the semantics, and it probably doesn't belong in ES spec, but it would be useful to have to point users to.

NM: for example we only lazily make `function.arguments` do crazy things, so that's not an easy thing to spec.

WH: do we really need new nondeterministic array methods? I agree that you don't want reduce operations that have serial semantics (add first element to second, the sum to the third, the sum to fourth, etc.) but could

deterministic tree operations (add elements pairwise, then add the pair sums, then add those in paris, etc.) be sufficient?

NM: the main one is `reduce`. You *can* do a tree-reduction, but that is not always the right thing to do on all architectures; it has a performance cost.

DH: also, we want to have developer tools give feedback on what can be parallelized; we are talking with a PhD student about working on this.

ARB: How does this interact with GC?

NM: Works well using nurseries.

AWS: A version of region based collection.

Newly Added Global Object

DH: Straw proposal `Symbol.iterator`, `Symbol.create`

AWS: A single method on Symbol, `Symbol.for('iterator')`

EA: You can have get and set and use set to register user defined known symbols.

MM: registration needs to be like interning

DH: We're only talking about location of these new things.

EA: Libraries want to register well known symbols

DH: This is not what LH is trying to address

MM: Built-in symbols, as specified by the spec: no mutation and no global channel

Well-Known Symbols available on the Symbol object itself

LH:

- well-known symbols
- Symbol

- Reflect
- System

LH: We should be designing the library system independent of modules

DH: (clarifies) Implementation dependency graph, can't rely on these new objects without knowing where they come from (modules)

LH: We need to decide, concretely, if these are exposed on the Global Object.

(RW clarifies for MM)

MM: Will this break the web?

LH: Conflicts are soft conflicts because the objects are configurable

LH: How will symbols be held?

```js
Symbol.iterator = @@iterator
Symbol.create = @@create
```

The value is the _initial_ value (as done with other items in the spec)

writable: false
configurable: true
enumerable: ?

AWB: need to discuss @@toPrimitive

RW: Does this really need to be exposed?

AWB: It's useful for any objects that have internal data (like Date)

Consensus/Resolution
- Global
- Symbol

- Reflect
- System

- Well-known symbols defined as properties of Symbol
- using given name, no "@@"
- https://people.mozilla.org/~jorendorff/es6-draft.html#table-6

Next Meeting:

Nov 19 - 21 at PayPal