Minutes of the: 45th meeting of Ecma TC39

in: Paris, France

on: 24-26 March 2015

1 Opening, welcome and roll call

1.1 Opening of the meeting (Mr. Neumann)

Mr. Neumann has welcomed the delegates at the Inria office in Paris, France.

Companies / organizations in attendance:

Mozilla, Google, Microsoft, Inria, jQuery, Facebook, Netflix,

1.2 Introduction of attendees

Jordan Harband (JHD) on phone, part-time

John Neumann

Allen Wirfs-Brock

Yehuda Katz

Domenic Denicola

Adam Klein

Andres Rossberg

Alan Schmitt

Brian Terlson

Sebastian Markbage

Jeff Morrison

Lee Byron

Erik Arvidsson

Dave Herman

Kevin Smith

Brendan Eich

Mark Miller

Jafar Husain

Istvan Sebestyen – on phone

Rick Waldron (RW) – on phone, part-time

1.3 Host facilities, local logistics

On behalf of Inria **Alan Schmitt** welcomed the delegates and explained the logistics.

2 Adoption of the agenda (2015/020-Rev.1)

The agenda was approved as posted on the github:

https://github.com/tc39/agendas/blob/master/2015/03.md

The following is a snapshot of the final agenda from Github:

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 (TODO: Ref document name)
3. Approval of the minutes from Jan 2015 (TODO: Ref document name)
4. ECMA-262 6th Edition
   - i. Feedback/Discussion on Release Candidate Drafts
   - ii. Last minute instructions to the editor?
   - iii. Motion to Approve Final Release Candidateand and refer to GA for their consideration and ratificaiton.
   - iv. ISO Fast track discussion
5. ECMA-402 2nd Edition
6. ECMA-262 7th Edition and beyond
   - i. Class property initializers (Jeff Morrison)
   - ii. Class decorators (Yehuda Katz)
   - iii. *ReverseIterable* interface (Lee Byron)
   - iv. A Declarative Alternative to toMethod (Allen Wirfs-Brock)
   - v. Additional Meta Properties for ES7 (Allen Wirfs-Brock)
   - vi. Function Bind and Private Fields Redux (Kevin Smith)
   - vii. Immutable Records, Tuples, Maps and Sets (Sebastian Markbage and Lee Byron)
   - viii. Compositional Functions (Jafar Husain)
   - ix. More export ___ from statements (Lee Byron)
   - x. 64-bit arithmetic update (Brendan Eich)
7. Test 262 Status
8. Discuss Future Status of ECMA-327 (Compact Profile) and ECMA-357 (E4X)
9. Report from the Ecma Secretariat
10. Date and place of the next meeting(s)
    - i. May 27 - 29, 2015 (San Francisco - Yahoo)
    - ii. July, 28 - 30, 2015 (Redmond, WA - Microsoft)
    - iii. September 22 - 24, 2015 (Portland, OR - jQuery)
    - iv. November 17 - 19, 2015 (San Jose - Paypal)
11. Group Work Sessions
12. Closure

3 Approval of minutes from January 2015 (2015/017)

The minutes were approved without modification.

4 Major administrative decisions to move "ES6 Suite" approval forward

TC39 decided the following:

4.1 RF "Opt out" for ES6 Release Candidate #1

Noting that it already started on February 23, 2015 – ending April 25, 2015

4.2 TC39 approval for ES6 Release Candidate #4

ES6 to be submitted to the Ecma June 17, 2015 GA for approval:

6 members in favour (Google, MS, Inria, Mozilla, Facebook, JQuery),

0 abstains,

0 opposed

Thus: Approved

ES6 to be fast-tracked to ISO/IEC JTC 1:

6 members in favour (Google, MS, Inria, Mozilla, Facebook, Netflix),

1 abstain (JQuery),

0 opposed

Thus: Approved

NOTE Netflix came into the room after the first part of the vote.

4.3 TC39 approval for ECMA-402 2nd Edition

ECMA-402 2nd Edition to be submitted to the Ecma June 17, 2015 GA for approval:

6 members in favour (Google, MS, Inria, Mozilla, Facebook, Netflix),

1 abstain (JQuery),

0 opposed

Thus: Approved

ECMA-402 2nd Edition to be fast-tracked to ISO/IEC JTC 1:

6 members in favour (Google, MS, Inria, Mozilla, Facebook, Netflix),

1 abstain (JQuery),

0 opposed

Thus: Approved

4.4 RF "Opt out" for ECMA-402 2nd Edition

TC39 decided to start the "opt-out" immediately, ending on May 26, 2015

4.5 RF "Opt out" for ECMA-404 1st Edition

There is no change on the in 2013 October approved standard. However, that was approved under the RAND Ecma patent policy regime. Since the JSON Syntax originally in ECMA-262 Ed. 5 has been moved from ES6 into ECMA-404 in order to assure the RF status of the entire ES6 Suite TC39 decided to launch an opt out on ECMA-404 immediately, ending on May 26, 2015.

NOTE 1 The fast-track option for ECMA-404 has been decided by the GA earlier on depending on the decision of TC39. TC39 decided that ECMA-404 (as all parts of the ES6 Suite) will be fast-tracked to ISO/IEC JTC 1.

NOTE 2 Both TC39 and JTC 1 are aware that the planned more frequent (yearly) update of the ECMAScript might create synchronization problems between the two standard versions. ISO/IEC JTC 1 is slower in its approval cycles. This is an open issue that needs to be solved.

NOTE 3 For the fast-tracked ES6 standards so called Explanatory Reports will be needed to provide additional information to JTC 1 members at fast-track voting. For ECMA-404 in draft this has been prepared. For the other parts of the ES6 Suite this needs to be done.

4.6 RF “Opt out” for ES6 Release Candidate #1

Noting that it already started on February 23, 2015 – ending April 25, 2015

4.7 ECMA-327 (Compact profile) and ECMA-357 (E4X) matters

Question: Should they be withdrawn?

ECMA-357 and ES6 will not work together.

TC39 is on the opinion that ECMA-327 and ECMA-357 should be withdrawn. **Mr. Sebestyen** will ask the opinion of all Ecma members, in order that final TC39 decision can be made at the next meeting.

5 ES7 and Test262 Discussions

Most time was spent to progress ES7 related topics.

For details please see Annex 1 in the Technical Notes.

6 Report from the Secretariat

Nothing in addition what has been said in connection with ongoing TC39 matters.

7 Date and place of the next meeting

I. May 27 - 29, 2015 (Bay Area, Netflix)

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

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

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

The meeting noted that from time to time a European TC39 meeting (like now the one in Paris) is useful and we should continue that practice. Possible next place might be in Munich (hosted by Google). This is just some thinking, not a decision.

8 Closure

Mr. Neumann thanked the TC39 meeting participants for their hard work. TC39 has reached an important new mail stone by finishing and approving ES6. Many thanks to **Mr. Wirfs-Brock**, the editor of ES6 for his hard work. Also many thanks to **Mr. Waldron**, the Editor of ECMA-402 2nd Edition.

Many thanks for the Technical note takers in Annex 1.

Many thanks to the host, **Inria** for the organization of the meeting and the excellent meeting facilities and dinner. Many thanks in particular to **Mr. Schmitt**. Many thanks also to **Ecma International** for the social event.

Annex 1

Technical Notes

March 24 2015 Meeting Notes

Brian Terlson (BT), Allen Wirfs-Brock (AWB), John Neumann (JN), Jeff Morrison (JM), Sebastian Markbage (SM), Yehuda Katz (YK), Dave Herman (DH), Alan Schmitt (AS), Lee Byron (LB), Domenic Denicola (DD), Kevin Smith (KS), Andreas Rossberg (ARB), Brendan Eich (BE), Erik Arvidsson (EA), Adam Klein (AK), Jordan Harband (JHD), Mark Miller (MM), Istvan Sebestyen (IS), Jafar Husain (JH), Rick Waldron (RW)

4 ECMA-262 6th Edition

(AWB)

AWB: (presents slides)

AWB: release candidates are intended to be complete subject to bugs so everyone could review them. RC1 was the review draft for the RF opt-out period.

AWB: bug count fluctuates toward zero and back up; minor technical or editorial issues. Waldemar found some small issues in the grammar (unnecessary parametrization, Unicode regexp grammar, ...). Bugs that come up between now and June will be fixed as practical, or deferred to ES7 if not.

AWB: any last minute issues!?

BE: what about the things that are backward-incompatible?

AWB: e.g. Brian has some issues around the function name property

BT: i.e., the assignment of names when assigning a function to a var declaration

AWB: EA has an issue around [[DefineOwnProperty]] on module namespace objects...

BE: built-in prototypes?

AWB: in RC1 it became pretty clear that at least making Array.prototype not-an-array would break at least some peoples' code. So for RC1 I reverted Array.prototype back to an Array exotic object. There was similar discussion about RegExp.prototype. For RC1 I did an experiment with some hacks that I thought would maybe give us a middle path. But that got reverted back.

BE: not sure there's any profit in trying to change it in the spec.

BT: that's what IE has: RegExp.prototype is an ordinary object; the only prototypes that are not ordinary objects are Function.prototype and Array.prototype.

BE: what about `Object.prototype.toString` on those prototypes? [Actually I was asking what RegExp.prototype.toString() returns. /be]

BT: it gives `[object Object]`. We haven't come across any code yet that breaks because of these changes, and to put that in perspective we have 3 or 4 sites that break due to the function name thing.

Module Namespace Exotic Objects

DH: I think there was a minor issue with the module namespace objects...

EA: my concern here is that we're introducing a new exotic object.

DH: that has been my intention for fully six years. (For some value of six.)

DD: I think the wiki said "`Object.create(null)` with getters"...

DH: yeah you're right, the exoticness is not the higher-order bit.

AWB: let me run through the things you can do with a MNS exotic object. A get; no set; a get with a symbol; enumerate; has-own (name or symbol); and that's pretty much it. Everything else fails (false return or throw). So e.g. Object.getOwnPropertyDescriptor will throw on one of these.

EA: a solution is to simply return a data property descriptor. Then code that uses Object.getOwnPropertyDescriptor, e.g. for mixins, will work.

AWB: the reason I made it throw is that the MOP lets you define things that are neither data properties nor accessors, and this is one of those cases, and the MOP lets you throw in those cases, so we could.

DH: it is self-evidently possible to write the spec as it's written now, but the question is whether we should.

MM: I would find it more useful to not throw.

AWB: even though it's not really writable?

YK: it *is* writable, but it's not writable by you.

AWB: I worry that we're establishing a precedent that we're going to make [[GetOwnProperty]] behave that way.

DH: there's a lot of code out there that has expectations about objects you give it, and if you build an object that breaks those expectations, then that particular class of object becomes rounded up and treated as toxic and people say to never use these objects, or alternately it starts to poison the well of these common operations, which is even worse. For example we did a similar thing with enumerate. Module objects ought to behave as close to normal objects as possible. It's a much narrower diversion to have an object which says it's writable but you can't write to it, than one that throws when you try to get a descriptor for its properties. On top of that, there is precedent for property writes quietly failing...

MM: you could write something with an accessor that has that behavior, but it's not normal... non-writable properties normally throw in strict, fail silently in sloppy.

AWB: being a data property means that if you get the descriptor you get the value of it. That means in [[GetOwnProperty]] I have to access the value, which could fail because of TDZ.

DH: this is the MOP. TDZ is part of the core semantics. How do we reflect TDZ in the MOP. We could throw, or we could have some special representation of TDZ state in the MOP (e.g. undefined). A throw seems better at this stage, in the case where you hit TDZ.

ARB: what happens when you do a [[Get]] in the TDZ?

DH: it throws

YK: why is it not just a getter

ARB/MM/AWB: then you have to reify the getter, and it's annoying to implement

Conclusion/Resolution

- Change module namespace exotic object [[GetOwnProperty]] to reflect as a writable data property.
- [[Set]] continues to throw in strict mode.

Function Name Issue

BT: we found several sites that are broken by the new function.name algorithm. One site we found is using function objects as maps, and so doing e.g. `if (func[key]) { ... }`. Since ES6 defines new semantics for `func.name`, this breaks the site. We have enough information such that we're not going to ship this. We could perhaps only infer for assignments to `let` and `const`, not `var`.

AWB: we infer lots of places...

BT: we've only seen problems for assignments to `var`.

MM: so if we removed it from `var` everything would work?

BT: yes, but it seems a bit hasty to make that change without testing...

MM: at what point in the process can changes like this not go in.

JN/AWB: we're there now.

DD: it's not a big deal if we have a different function.name algorithm in ES7.

DH: no, really!? If we have something that pretty clearly needs to be fixed, we should be able to change right up until the GA votes.

AWB: the GA needs the time to review it...

DH: in a 1300-page spec, a small change like this is not going to change anyone's opinion about ES6.

AWB: it's a process thing. This is about a feature, and it's either in or not in.

DH: the feature is in. I'm saying this is at the level of bugfix.

ARB: in Chrome every function has a name property....

BT: the problem is that it switches from falsy to truthy when it becomes a non-empty string.

BE: why are we changing anonymous function names?

MM: we're not changing anonymous function names, we're changing which functions are anonymous :)

ARB: which is one of these webpages?

BT: cheezburger.com, the comments won't load.

BE: stop ship bug, then!

ARB: cheeseburger.com?

AK: no, cheezburger.com.

DH: we should at least have a GitHub errata page where we can enumerate these things...

DD: once the great tooling revolution arrives, there will be a live GitHub version of the spec that implementers consult.

DH: yeah, but until the Great Leap Forward arrives, we need something in place.

YK: let's put errata in tc39/ecma262.

MM: having this be an errata-managed issue seems fine to me.

BE: what about a new property, instead of .name?

AWB: on Twitter somebody suggested it should be a symbol

MM: we're standardizing the de-facto precedent. The reasonable backoff suggested by the data is to not infer the data from the var assignment.

DD: two options. Make the change now with no var; or, leave the spec as is, let Microsoft accumulate more data to see if no-var-assignment is sufficient; if so then use errata to officially recommend those.

MM: let the ES6 spec and Microsoft's implementation diverge, and use that to figure out exactly what the right fix is.

EA: we should either do full inferencing or revert to ES5 semantics.

MM: what about concise methods?

EA: that's fine; they're new syntax

AWB: removing name inference would be a big deal.

BT: there are going to be more issues like this; errata management seems like the way to go.

AWB: indeed, the nature of the process is that we don't get this level of information until we're at this point in the process, with implementations.

BT: honestly, this issue is small compared to sloppy mode const semantics, which we've found issues with as well.

(general consternation)

MM: did we have a resolution for the class bindings issue raised on list?

AWB: probably leave as status quo...

DD: what about making them const?

AWB: that was the other possibility...

YK: there was a reason we didn't do that... I remember feeling strongly at the time.

DH: global?

YK: in the global space, people want to replace things that are global.

MM: I retreat...

AWB: OK, so, I don't think we should keep fishing on "is there anything else we should change"?

Conclusion/Resolution (for function.name issue)

- Leave the spec as-is
- Let Microsoft experiment with more web-compatible versions of the algorithm, e.g. not-for-`var`
- Have a public place to track and post errata regarding this issue, until the Great Tooling Revolution of 2015 arrives

Let's Ship This Thing

JN: we need two votes; we vote to refer the spec to ourselves, then we vote to approve the spec.

(general confusion, slowly turning to incredulity and then acceptance)

AWB: (presents slide titled "A Motion")

DH: (joking) I filed a patent for the Window object. It is a very unique design.

MM: (joking) if you get it granted, *don't license it to anybody*.

AWB: we intend to approve the final draft here, but conditional upon completion of the RF opt-out period.

JN: (conducts a vote)

the TC39 RFTG has approved referring ES2015 to the TC39 TC (unanimous)

the TC39 TC has approved referring ES2015 to the Ecma GA (unanimous: Google, Microsoft, Mozilla, jQuery, Facebook, Inria)

JN: one last vote! should we recommend to the GA that they refer ES2015 to the GA for fast-track approval. During ES5, the Japanese national body of the ISO recommended a large number of changes which we incorporated as ES5.1. If we refer this document to ISO for fast-track, then we can assume Japan and others will make comments. Those comments will be approved.

AWB: this is the complication of this fast-track stuff. "Fast" means "about a year." If we start this down the fast-track, it will interfere with ES7; about the time we exit the fast-track process for ES6, we will also have ES7.

MM: we do need to figure out how to get this working, but the entire thing is worth it if for no other reason than the high-quality contributions of that one reviewer at the ISO. I don't know how else to engage him.

(discussion of new text vs. old text, will reviewers look at the deltas)

DD: there may be sections that are untouched, but they are very few.

AWB: there may be sentences that are untouched...

IS: it would be very unkind not to refer this to them. My suggestion would be that we should go ahead for the fast-track but it should be a conditional go-ahead on myself and John sitting down with ISO management and discussing a way to find a solution to integrate the different cycles. It would be ugly if there was a large delta between ISO's version (~ES5) and ours.

JN: it exacerbates the system right now. We have a four-year window between ES5 and ES6 and the plan is to have one-year windows. ISO may choose to wait until ES8 or ES9 to fast-track, bypassing ES7. They want to pick up the massive changes between ES5 and ES6. So I think it's important to do it this time and less important to do it next time.

DD: couldn't we just do this because it's good but not spend much time worrying about it?

AWB: unfortunately it's not that simple. The manpower to publish documents is not free; it would take away resources from feature work etc.

MM: Allen what's your opinion?

AWB: I think we need to do ISO. There is significance involved in that.

MM: so going forward with our new process do we do ISO every year?

AWB: I don't know, I only did it once and it took a year.

JN: it could take a longer this time, fast-track has changed.

AWB: should we do a partition standard?

JN: we don't want to go down that road...

AWB: maybe it would help make this maintainable over time?

JN: I'm not opposed, but it needs to be thought out...

AWB: maybe a partition standard could be moved through the ISO process independently...

IS: there should be minimal market confusion because most people download the Ecma version, not the ISO version...

DD: most people download the version on Jason Orendorff's personal web page...

IS: download counts support many more Ecma downloads than ISO downloads ("much much much much smaller")

AWB: that is probably why it is hard to get people in this room excited about ISO...

JN: so I think what we want to do is fast-track this year, but future fast-tracks would be questionable.

JN: so, let's vote (per company): do we approve submission to ISO for fast-track.

approve: Google, Microsoft, Mozilla, jQuery, Facebook, Inria, Netflix

opposed: (none)

abstain: jQuery

IS: ECMA 402 can be an ECMA-only standard according to Patrick. ECMA-only standards can be used as normative standards, so it's up to us whether we want to fast-track. Doesn't seem critical. ECMA-404 is the more interesting one.

AWB: Rick isn't here - are we also going to approve 402 right now?

YK: Is 402 aligned with ES6?

AWB: I hope so, ES6 normatively references it.

YK: Will ECMA be upset if we normatively reference a non-normative spec?

AWB: Yes.

AWB: Istvan, do we need a opt-out period for ECMA-402.

IS: Yes.

AWB: My understanding is that Rick has created the final draft and it has been available for a month for people to review.

IS: It would be nice if we could approve at this meeting, as far as I know this was the intention of the editor. So seems like we have a small problem.

AWB: Check with Rick. If people here take the time to look at the draft, by the end of the week we could vote to refer 402 to GA.

IS: We can't reference 402 edition 1 from ES6?

AWB: No.

Conclusion/Resolution

- Everyone to review or do what is necessary to vote to pass along 402 on Thursday.

What to do with compact profile (ECMA-327) and E4X (ECMA-357)

AWB: These are still active standards. E4X still gets downloads. We can withdraw the standards.

BE: ActionScript 3 in Flash implements E4X.

IS: We had this on the agenda when we did ES5. At that time the point was that even if it was out of date, if implementers wanted to implement the old one they could. Adobe may have wanted this. 357 is often downloaded. But I don't know how serious this is.

AWB: E4X is not aligned with ES6 so it seems misleading for it to be out there.

IS: Some people may have used it with edition 3.

AWB: We had this conversation 5 years ago. Things have evolved. Maybe adobe still cares, or maybe they don't.

BE: It can be withdrawn without breaking other standards.

IS: We approved ECMA404 before we had the RF agreement. Should we start an opt-out period for ECMA404?

Conclusion/Resolution

- Istvan to follow up with Adobe and the CC to see if they would object to withdrawing E4X

ECMA404 RF?

IS: We approved ECMA404 before we had the RF agreement. Should we start an opt-out period for ECMA404?

AWB: Can we retroactively make ECMA-404 Royalty Free?

IS: Yes

AWB: We should have an opt-out period then.

Conclusion/Resolution

- Start an opt-out period.

Compact profile

DH: People who are interested in this can start doing work. Once there is actual demand work will happen. We shouldn't just make standards because someone might use it.

AWB: I think there is consensus to kill 327.

Conclusion/Resolution

- Istvan to follow up with Adobe and the CC to see if they would object to withdrawing the compact profile.

Fast track ECMA-404

IS: The actual standard document is not ready, maybe ready for this afternooon. Then we can publish it for fast track.

Conclusion/Resolution

- Revisit tomorrow.

6.1 ES6 Class Properties

(Jeff Morrison)

https://gist.github.com/jeffmo/054df782c05639da2adb

```
class Counter extends ReactComponent {
  constructor(props) {
    super(props);
    state = {count: this.props.initialCount};
  }
}
Counter.propTypes = {initialCount: React.PropTypes.numer};
```

JM: React ends up with some boiler plate code because it wants to add an own instance property.

```
class Counter extends ReactComponent {
  static propTypes = {initialCount: React.PropTypes.numer};
  state = {count: this.props.initialCount};
```

JM: Issues with storing instance values on the prototype. Reference types shared between instances.

JM: Transpose initializer into the constructor has issues with the scope.

JM: Wrap each initializer in its own closure from within the class-body environment.

JM: Store the closure as an 'initializer' value in a property descriptor on the prototype.

MM: Hold on. Adding another field in a property descriptor is very heavy handed and has to go through the whole proxy design.

YK: This is needed by other proposals. Not in the MOP though. Opposed to add new features that are not reflected.

AWB: Need to define when this happens. End of `super()`?

JM: Wants to move passed where the initializer is stored.

JM: Last step of construct. Immediately after calling super()

AWB: Inconsistent. In a non derived you cannot access the own instance field.

JM: That is by design. Sorry, it is the other way around. The initializer is executed in the base class before the constructor body is entered.

AWB: It is important to create properties up front (with an unitialized state or undefined).

YK: It is important to put these on the prototype because it allows reflection.

AWB: By putting these on the prototype they are visible to the application. If these have privileges then those privileges will leak.

DH: Not convinced that this is important. The important part is that we need to store the initializer behavior somewhere.

MM: Is it configurable?

DH: Not fundamental. JS assignment creates the configurable writable enumerable.

DH: With decorators one can create non writable etc.

MM: I would think that non configurable would be good so that the shape is guaranteed.

DH: For backwards compat we should stick with the old behavior.

AWB: This is about creating the instance shape. What new thing is this getting into. It provides, prior to construction, information about the shape.

DH: When we provide new syntax it is tempting to alter the semantics instead of following the patterns used today.

AWB: It is possible that the shape is all about private state and this is all about classic prototype oop.

MM: Using a descriptor is very heavy handed.

YK: I want it to be pervasive so a descriptor is good.

MM: If you put it in the descriptor.

FM: You can use any object as a prototype.

YK: And if a class constructor uses that as a prototype then you can get own instance properties.

AWB: What are the implications if every descriptor needs to have this new property?

AWB: Are you assuming that the current design with property descriptor is a good model to extend for reflection mechanism.

MM: The concern I have is that there is a property with the same name on the prototype but its purpose is not the same.

MM: What you want is something whose value is the initializer function.

DH: What you want is some association with the name and the initializer function

MM: What you want is that your initializer function returns a descriptor. Then the decorator can operate on that descriptor.

DH: Some way to associate the initializer. You can use a symbol.

DD: The decorators already act differently. prototype or on the constructor.

YK: Wants to operator on more than the value. Same way for methods as for inst props.

DH: API like method decorators.

```
sidetable.myinstprop = {value: func () {}, configurable: ..., ...};
```

YK: It is important to distinguish these from value properties. For example there migh be a use for creating own instance methods. Therefore using initializer instead of value is important.

AWB: When I create lots of objects I don't want to have to pay for all these internal MOP.

YK: The reified methods only come into play if you use reflection and decorators.

AWB: What if someone modifies these descriptors.

EA: It is important that the descriptor of these own instance properties is immutable, or at that you cannot change these afterwards because then the constructor function has to change its behavior after the fact.

Conclusion/Resolution

- Stage 0
- Do not use descriptors for initializer state.

6.2 Decorators

```
class {
    @readonly name() { }
}

function readonly(prototype, name, desc) {
     desc.writable = false;
     return desc;
}
Object.defineProperty(Person.prototype, 'name',
  readonly(Person.prototype, 'name', {
    enumerable: false,
    configurable: true,
    writable: true,
    value: <closure>
}));
```

MM: When defineProperty runs, does Person.prototype have an undecorated name property?

YK: No. The thing that you pass to the decorator looks like a descriptor, but is just an object.

AWB: We need to think about what the built in ones should be.

YK: Things that update descriptors seem like good candidates.

http://bit.ly/ember-js-classes

Examples of things that Ember might do with decorators.

YK: Shows example of more complex decorator usage:

```
class Person {
    firstName = "…";
    lastName = "…";
    @concat('firstName', 'lastName', ' ') fullName;
    @concat('lastName', 'firstName', ', ') formalName; }
```

MM: Can create a read-only getter from a declared property

AWB: Do you have a more plausible use case for more complex decorators

YK: (Shows ember gist. Shows @service which does dependency injection.)

AWB: Why would I use a decorator instead of a property initializer?

YK: More complex

JM: Decorators run once per class

AWB: This is an own property?

YK: The decorator would attach a more complex getter. @service does a lazy lookup.

DD: When you return a descriptor does it return an own property?

YK: ?

MM: Since this is integrated into TypeScript, how does TypeScript type the decorator method?

YK: Another way to ask question is I would prefer a more immutable API so that I know what the type is. Decorators themselves are typed, but they can mutate.

MM: It's a functional transformation of a descriptor. There's nothing being mutated on.

ARB: This is reified as a first class value. How can you track that without a full blown dependent type system?

YK: (Shows examples of decorators with no mutating effect).

DH: ARB is saying that the set of properties that end of being in the class ends up depending on what happens with these decorators.
Mutation is a red herring. The issue is computably determining the class type. You're saying that there is an expressive subset of this system for which this is not a problem.

ARB: To know what to reject you'd have to know the returned value at type-check time.

JM: ARB is saying that the value is dependent on runtime. Enumerable true and enumerable false produce very different things.

ARB: Hard to forsee the implications, because annotations will want to spread.

YK: We should consider them on a case-by-case basis, but I am conservative. I'm not proposing targeting general declarations.
Use case is that with class syntax we aren't able to do things we could with object literals.

MM: Are you proposing that you can decorate fields in an object literal?

YK: I don't consider it fundamental but there is a symmetry concern.

ARB: Still missing the big picture. Don't you ever want to decorate parameters and other things?

YK: These are things we might want to consider.

ARB: Take modules, it seems like people will want to do some metaprogramming by annotating imports.

YK: Proposal is addressing an expressive hole introduced by ES6. There is not a way to do higher order things with methods.

DH: We're targeting the class syntax but we need to focus on the whole and not the part. The broad category is metaprogramming. What is our model?
In Scheme, it's static. In Ruby, it's dynamic. We should do a zoom out and look at what metaprogramming might mean for the language.

ARB: (Objects to saying that JS implies dynamic metaprogramming.)

DH: (Mentions objection to staging.) With a first class class system, a lot of meta- programming stuff you just do at runtime. If you've gone down the path of dynamic metaprogramming, then it's probably best to keep going that way. ARB: I think class expressions are a red herring.

DH: I just don't believe that we should do staging.

BE: staging not seriously proposed but noted as possible accidental outcome of generalizing decorators across first- and second-class parts of ES6. We want a principled approach that's not just for classes.

AWB: Say you had a decorator that was enumerable or non-enumerable...

YK: Users will want to apply descriptor decorators to object literals as well.

YK: Don't see why we should restrict class features to static

DH: Javascript is fundamentally a dynamic language and we shouldn't try to make it static

ARB: The dynamic stuff will remain, but we'll have these conflicts.

ARB: Original question was what problem are we solving?

DH: Need a more general system for "twiddling" knobs.

ARB: My question is how does this generalize?

DH: Current instinct is that first class things (classes, object literals) are easy to apply this to, but for second class things we'll fall off a complexity cliff. Agree that we should look at the bigger picture.

ARB: readonly is an example that we might want to have a more direct way.

YK: If readonly were built-in and you imported it, then the engine could do smarter things.

DH: Suggests stage 1, but audit the space which may lead to changes.

Conclusion

- Stage 1, YK will audit the space.

6.3 ReverseIterable interface

(Lee Byron)

Proposal: https://github.com/leebyron/ecmascript-reverse-iterator

LB: presents slides

[lots of discussion about how to extend iterators/iterables]

Conclusion

- More work to be done in breakouts to figure out path forward for iteration.

March 25 2015 Meeting Notes

Brian Terlson (BT), Allen Wirfs-Brock (AWB), John Neumann (JN), Jeff Morrison (JM), Sebastian Markbage (SM), Yehuda Katz (YK), Dave Herman (DH), Alan Schmitt (AS), Lee Byron (LB), Domenic Denicola (DD), Kevin Smith (KS), Andreas Rossberg (ARB), Brendan Eich (BE), Erik Arvidsson (EA), Adam Klein (AK), Jordan Harband (JHD), Mark Miller (MM), Istvan Sebestyen (IS), Jafar Husain (JH), Rick Waldron (RW)

6(iv) A Declarative Alternative to toMethod (Allen Wirfs-Brock)

AWB presents https://github.com/allenwb/ESideas/blob/master/dcltomethod.md.

Object.assign with 2nd arg object literal has a `super` hazard.

`toMethod` was previous workaround, but easy to forget and hard to use with object literals.

YK notes `toMethod` still valuable addition to meta-programming API.

AWB detials further issue of deep-clone vs. shallow-, unresolved (esp. viz `.prototype`).

Discussion of whether we can avoid `toMethod` entirely, not just for this use-case. MM hopes so, YK dashes hope by assertion -- to be continued later.

AWB observes that object literals and classes handle `super` and other contextual forms fine, by being special forms affording sound [[HomeObject]] initialization opportunity. Therefore proposes `mixin` contextual-keyword operator to extend object literal special form for this use-case.

NB: `mixin` is postfix operator with what looks like an object literal after it -- not binary operator.

MM: I suggested to AWB that the operator should be named `mixin=`, so it resembles the assignment operators (`+=`, `*=`, etc.).

All object literal syntax on right of `mixin` is allowed, *except* for `__proto__`.

`mixin` does [[DefineOwnProperty]] based on object literal contents (unlike `Object.assign` which uses [[Set]]).

Name abstraction of object literal for later param to `Object.assign` (still permitted, a bug if `super` used) doesn't work:

```
    let mixins = {...}; Object.assign(target, mixins); // how to do this with
mixin operator?
```

Use an arrow:

```
    let mixins = obj => obj mixin {...}; mixins(target);
```

Some inconveniences with classes (see URL). Solution is `mixin class {...}` extended special form. Throws if left-hand side is not a constructor (spec `IsConstructor` test returns false). Class body to right cannot include `constructor`. Gets non-enumerability of methods, static methods, right.

DH: question of user expectation of what's expected "moving" method from object to object. `this` as dynamically bound, lexically bound for arrows, understood; ditto lexically scoped upvars. Supposes we would have preferred `super` to be implicit parameter, akin to `this`, but we didn't do that. JS made method extraction easy, so it's common -- `super` as distinct from `this` goes against grain. So extracting a method with rebound `super` still wants `toMethod` -- `mixin` doesn't help.

AWB: "if we had been braver, maybe we could have made `super` be dynamically bound... but we didn't." (supposes JITs would have optimized away unnecessary super-params) DH: right, too late -- and who knows if it would have worked... but we cannot dismiss `toMethod` use-case for single-method extraction, even with cloning issues. DD/EA: agree, cannot desugar to existing functions not expressed as method of `mixin` right-part literal forms.

MM: no matter what we do, we can't make ES5-ish method-extracting code continue to work with `super` added somewhere in the method body DH: This is a regression of expressiveness. MM: Old library code that does mixins the old way is ok if used only in ES5-ish way. YK: If `toMethod` available, people will write patches to fix such library code.

Group debates exact method-extraction expressiveness/safety regression, mourns the loss. Some general sense that we should not throw `toMethod` baby out with bath-water.

ARB: this proposal is very imperative. If we want provide high-level support for mixins, especially with suggestive syntax, then it should be declarative, e.g. like traits. e.g. Scala `class Foo extends Bar with Baz...` or early ES6 class proposals DH: real problem with syntax that hides mutation, misuses `mixin`. AWB: can bikeshed operator name. DD: `mixin` is already variously defined by ecosystem, wrong word here.

MM: This is mid-level abstraction, should we do it or provide only high-level traits as ARB suggests? plus `toMethod` as low-level -- if high- and low- without mid-, would we need this mid-level proposal? AWB: "a lot of stuff in JS happens at mid-level." MM: if high-level needs language extension, doubts mid-level; if mid-level enables self-hosted high-level, may be ok.

DD: if `mixin` misnomer not used, then this would probably be uncontroversial. (BE: not uncontroversial with ARB) DH: been down design path of fake object literals (with triangle), hard to avoid kooky outcomes. DD: really need some syntax for [[DefineOwnProperty]], must avoid connoting "assignment" that runs setters. MM: if we used `:= value` then value would not be property descriptor, so `:=` ain't right either.

BE: beware justifying more kludges because JS is kludgey. There i said it!

MM: how do you write the trait to be mixed in? Lambda-abstracted form (obj => obj mixin {...}) pleasant. AWB: dares to write `:={...}` instead of `mixin {...}`, DH calls it the barkeep operator. :={

DH: is this about 1. a collection of properties from which 2. to mutate the target? Need something connoting that double meaning.

YK: re: MM's lambda-abstracted point, can linearize mixins along prototype chain and get `super` chaining to work. YK shows Ember's `Mixin.create` example (LINK NEEDED) demonstrating this. Distinct from AWB's proposal, more about traits in JS than about define-properties-on-target mid-level. DD: back when Chrome Canary had toMethod, I was able to use it to create these kind of mixin-proto-chain things: http://jsbin.com/fepudi/2/edit?html

YK: my Ember-based lambda-abstracted class-extends expression is a bit too much mechanism BE: or ceremony? YK: see https://gist.github.com/wycats/f79fe019d4bf29177b6c part-way down. MM: I like it, what's the problem? YK: wouldn't it be better to support high-level mixin syntax:

```
mixin TextSupport {...}
class MyComponent extends EmberComponent with TextSupport {...}
```

MM: don't need more syntax YK: indeed, could just have

```
class MyComponent extends mixin(EmberComponent, TextSupport) {...}
```

YK: I'm personally ok with this.

EA/DH: want happy-path "blessed syntax". DH: but let's not rush to design it here!

MM: with arrow-based lambda-abstraction + YK's `class extends` chaining for linearized `super`-preserving mixins, I don't want `mixin`. Chaining multiple `super` calls is winning here.

AWB: still leaves a mid-level gap, for define-these-properties-on-this-target. MM: is cost of plugging this hole worth the benefit, given new syntax very-high costs as noted? AWB: we need a stage 0 proposal for at least some two of three {low,mid,high}-level ideas.

BE: did we not agree that low-level `toMethod` is needed. Group: not in light of new news.

DH: advocates `toMethod` as analogous to `bind` AWB: not so, clone issue MM: not so, more important: `bind` does not allow `this` rebinding, `toMethod` allows generating new `super` bindings in new cloned methods If you could write function with free `super` that would throw without `toMethod`, then ok -- but we don't have that If you were constrained by new syntax to write a factory of super-bound methods, then ok too. YK: some extra boilerplate.

AWB: want to use concise method syntax since it handles super correctly, somehow thus the idea of `target mixin { method() {...} }`.

MM: difference between method-with-unbound-super and lamba-abstracted class-extends mixin/trait chaining: latter does not imply `toMethod` plausible to have low- and high-level that cover the space without AWB's `mixin` mid-level. MM: kind of want function-bearing-unbound-super to be not callable. AWB/YK/BE: function* vs. function precedent, but don't want more sigils/function-suffix-punctuators

MM: sketch of special form that allows unbound `super`:

```
function (super) foo (x, y) { ...super ...x ...y }
```

Factory of methods with unbound `super` uses. Trying for low-level principled alternative to `toMethod`.

AWB: alternative I prefer would leave function with unbound `super` uncallable, add `Reflect.toMethod` API for people to build libraries. BE: syntax for low-level is unusual, really prefer API high-level as ARB suggested (Scala) would be new syntax MM: lambda-abstracted class-extends relieves me of wanting anything like that high-form BE: true, although new users and mass-market programmers still "WAT" at the boilerplate but it's not that bad here (vs., e.g., the module pattern)

KS: painful if unbound-`super` in functions allowed, because define-properties-on-target requires long-hand unconcise-method syntax ARB: why not allow `function f(super, a, b) {...}` and (in concise method position) `m(super, x, y) {...}` instead? MM: Oh! YK: probably few can track our discussion at this point!

YK: want to avoid "harsh end of life" outcome for ES5-ish libraries.

KS: if we had not bound `super` in concise methods in object literals, then we could use concise methods without wrong-super fear. BE: is ARB's leading `super` parameter dead? YK: it's easy to forget and kind of weird DH: It's like default-final, you have to opt out too much and people forget.

AWB: need some stage 0 proposals, at two of three levels. YK: not sure we want Babel implementing this yet. YK: Babel should not implement stage 0, wait for stage 1 or later. Actually use stage-N as flag. EA: Babel can do whatever they want DH: We should care, because ecosystem effects, premature de-facto standards

Agreement that Babel (and other compilers) need to flag early-stage stuff, not expose prematurely, message well.

6(v) Additional Meta-Properties in ES7 (Allen Wirfs-Brock)

(https://github.com/allenwb/ESideas/blob/master/ES7MetaProps.md)

AWB: [presents slides (link?)]

AWB: function.callee: refers to the currently-executing function

MM: What's the use case for `function.callee`?

AWB: Referring to anonymous functions, or arrow functions

MM: Just assign it a binding in the containing scope. Say, a let binding.

AWB: Lots of requests to support callee, even though there are these other ways of supporting it.

JH: New syntax should start "10000 points down"

MM: Possible use case: referring to a concise method from inside the method's body

DH: Or inline event handlers in HTML

YK: I really hope people don't use it in that case

EA: Even for addEventListener, some people want this, since it's common to just pass the function as an argument without naming it

BE: You could have concise methods bring their name into scope, but that runs afoil of the same issue with dual class bindings in ES6 (immutable in the class body, mutable in the containing scope)

DH: There seems to be a disconnect between this committee's confusion over why anyone would need this feature and emphatic requests from the community for its inclusion

YK: The biggest use-case is callback-based APIs where the function needs to use its own identity to refer to itself, say to call `removeEventListener`

DH: That sounds like the best argument so far. Maybe the problem is the name? It sounds fancy, but it's statically always *this function*

MM: How does `function.callee` interact with arrow functions? It seems like it must refer to the enclosing function, just as `this` does

DH: Agree

DD: Disagree

DH: Explaining why arrow functions are treated specially: arrow is not syntactic sugar for `function`, they are a different kind of thing, as close to TCP as possible, unlike concise methods, which are "close to" sugar over `name: function() {}`

DD: But concise methods are not just sugar: see super bindings

YK: People who have tried to replace all functions with arrow functions find that they do not behave the same

MM: The analogy for arrow functions is to blocks, not functions

JH: Is "block" really a good metaphor, given that ES5 did not have any block scoping?

MM: That's the best I've been able to come up with

YK: It's really close to blocks in Ruby

BE: "function" is much too long. Maybe there's another keyword we could use? "function" is both too long, and confusing due to arrows.

DH: Do we even want this to refer to arrow functions, syntax aside? I say no.

DD: But if the main use case is event listeners, then that doesn't help

DH: Major problem is that referring to the arrow function is a refactoring hazard (e.g., moving it inside a forEach will result in a different answer). But maybe the argument is that it's a reflection feature, and reflection features are already refactoring hazards.

YK: There are two kinds of callback-based APIs in JS: synchronous and asynchronous. In the sync case, you want TCP. But for the async case, you don't.

JM: The explanation to the average programmer can't be "because TCP"; people won't understand that

MM: Before arrow functions, programmers constantly used `this` inside a callback and expected it to refer to the `this` of the containing function. The reason I explain arrow functions as blocks is that it's just "I want to run this code as a callback", with all bindings identical.

JM: I think the problem is just that I've tried to explain TCP to people

YK: Don't try to explain it that way. Maybe the problem is that we only have fat-arrow (designed for the synchronous callback case), but people want something that also works in the async case (which could have been thin-arrow).

DD: Rethinking, due to the forEach case. Maybe I do want `function.callee` to refer to the outer function.

YK: I think we need two things: one that refers to "this function" (including arrows), and one that does not include arrows.

ARB: Where does that end? Why not add more levels of function?

YK: We already have these two levels that have different behavior (arrow and non-arrow)

DD: Is it fair to say the argument is between 0 and 2 ways of refererring to a callee?

MM: Do we have any quantification of the need for this field?

YK: It used to be a big deal for me in ES5, but I got over it

DD: My problem with having 2 ways is it adds a cognitive burden whenever I want to use this thing.

YK: I agree that it's a cognitive burden to have 2, and that that may point towards having 0

AWB: I'd make an argument for 1: it refers to the innermost function, whatever kind of function it is (including arrows)

MM: This whole discussion is the result of an anti-pattern, that there's a requirement to refer to function identity to interact with, e.g., `addEventListener`, instead of having it return a token

YK: `addEventListener` is a void function, so it could be fixed in DOM

DH: Someone would have to do that

DD: setTimeout already returns a token, and in Node it's the right thing, rather than a number like on the web

MM: Caja actually wraps setTimeout and has never run into compatibility problems (jQuery works fine)

...moving on with AWB's slides...

AWB: function.count to find out how many arguments were actually passed.

ARB: Why are we working on this now? We don't have much experience with how people are using ES6 in the wild.

DH: Now is a good time to explore this space. In strict mode, we got away from using magically-scoped variables for introspecting on certain kinds of things. `new.target` provides the blueprint for a new way to expose these things, so it seems reasonable to consider adding such things with this new syntax.

AWB: ES6 provides lots of new ways to pass arguments, but doesn't provide an easy way to answer the question "how many arguments were passed"? This is useful for overloading based on the number of arguments. Using `...args` and destructuring is a pain.

DH: Why not `function.length`?

AWB: To avoid confusion with the `length` property of Function instances

ARB: Don't we have the same issue with arrows again?

AWB: Yes.

DH: We could make each decision at a local maxima, and end up with a completely confusing set of cases. Another option would be to have `function.callee` be a record, with other properties hanging off of it to ensure consistency between the different properties.

DH: The two concepts are "nearest enclosing function" and "nearest enclosing callable thing"

AWB: next up, `function.arguments`: the actual argument values

MM: How is this different from `arguments`

AWB: You get a fresh array each time you ask for it

DH: Why?

DD: It could be a frozen array and always return the same one

MM: I still don't understand what advantage this has over `arguments`?

AWB: It's a real array

YK: Use case is to name individual arguments, and then pass `...args` to some other function

MM: But why couldn't you use `arguments` for that case?

AWB: The difference is that it's a real array. Also, we're walking the line of whether `arguments` is considered deprecated.

DD: And it works in arrow functions.

DH: I agree with AWB's vision that `arguments` is deprecated, and that's the argument that makes `function.arguments` make sense

MM: What is the evil of the strict arguments object such that we would want to deprecated that and not deprecate this?

DH: Strict arguments is TCP-violating

MM: So is `function.arguments`

DH: I admit it's not a strong argument

JH: How bad is using `...args` and having to use destructuring on the next line?

EA: You lose documentation of what the actual arguments are expected to be.

DH: Options are you use one of 3: 1. ...args and destructuring (bad for documentation) 2. named params and ...args (have to do math) 3. `arguments`

AWB: But won't `arguments.length` materialize the arguments object?

BE: Not in Spidermonkey, or in V8

YK: If we're going to go with `arguments`, then we need to stop saying it's deprecated

DH: So which parts of `arguments` are still deprecated? We all agree that `arguments.callee`, `arguments.caller`, and `Function.arguments` are deprecated.

YK: Indexing `arguments` still seems like a bad practice in ES6. Spread is OK, though.

DH: That is not a coherent position.

YK: The thing that's not OK is treating `arguments` as first-class value

DH: That seems like a very fine distinction

DD: Can we pop the stack here?

AK: The question about whether we can do indexing seems separable from whether to call the array-like thing `arguments` or `function.arguments`

DH: It's important to get clear on what to say about `function.arguments`, and what to say about deprecation

BE: I think it's fair to say we don't want to spend all afternoon on this

...break...

Initial Value Passed to first call of a Generator next function

```
function * gen(a)  {

}

var g = gen(1); // this?
g.next(1); // this?
```

AWB: `function.next`: the current yield result

MM: Of all of these, this is the one I find compelling

AWB: These don't have to all be in a package. We can do one of these without the others

MM: I think we should do that with this proposal and move it through the process

YK: Can someone state the compelling use-case?

JH: A good usecase is a lexer. You need to get access to the first character that was passed to `next()`.

...next slide...

AWB: `function.thisGenerator`: current generator instance

MM: If you want this, you can wrap your generator function with another function, bind the generator in the function scope, and then you can refer to that from within the generator function.

YK: But what's the use case?

MM: That's a separate question, as the above suggestion allows referring to the generator if there are use cases

BE: Is this something people asked for? It doesn't seem that important.

...next slide...

AWB: `export.name`: current module identifier

DH: Do we need `export.name` if we have `import from this`?

DD: I thought it was `import from this module`

DH: Don't like compound keywords

YK: I'm still opposed to `import from this`.

AWB: Other proposals: `import.this`, `import.meta`

DH: Let's discuss separately.

...end of slide deck...

AWB: So I'd like to put all these at stage 0.

MM: I propose we make `function.next` stage 0 and leave the rest.

DD: I got the impression that we rejected the rest.

AWB: That's not my impression

MM: What does stage 0 require?

DD: Only being not categorically rejected

[discussion of how to handle organization of stage 0 proposals; how to make it clearer which stage 0 things we'll do and which we won't]

MM: So can we list these as rejected proposals and bring them back later if we change our mind?

AWB: I think that would be a mistake. That seems too strong for these.

Conclusion/ Resolution

- `function.next` to be split out into its own proposal
- AWB to keep the rest alive for now at stage 0

Report on 402

(Rick Waldron)

RW: Complete, barring any editorial bugs.

- Has been reviewed by Norbert Lindenberg and Andrée Bargul.
- Assembled a whole team to read the extensions to the spec.
- Added 402 repo to Github. With same proposal pipeline as 262.

Making a motion for acceptance/ratification/whatnot of 402 2nd edition... forward to the ecma assembly, subject to the rf opt out...

Unanimous approval to forward this to TC39 rftg.

TC39 vote 6 in favor. 1 abstains (JH, Netflix)

IS: Shall 402 be submitted for ISO fast-track? Or leave it ECMA-only like the first edition?

JN: Send it to ISO and see what they say

Conclusion/ Resolution

- Submit to Ecma GA
- Submit to ISO

6(vi) Function Bind and Private Fields Redux (Kevin Smith)

https://github.com/zenparsing/es-function-bind https://github.com/zenparsing/es-private-fields

KS: [presents slides (link?)]

KS: Abstract references recap.

KS: Problems with combining the various use cases. Instead, provide two different syntaxes, one for function binding and one for private state.

KS: Part One: Function Bind via `::`

```
function f() { return this.x }
let bound = ({ x: 100 })::f;
bound(); // -> 100
```

DH: This could be really nice in that you don't have to use `bind` anymore

KS: `::` doesn't take care of all of `bind`: only binds `this`, not any arguments

ARB: Syntax somewhat in conflict with `::` in C++, inverts the meaning of '::' vs '.'

AWB: Prior to ES6 there was a lot of confusion about what `this` means. With ES6 classes (and arrow functions) we made the story a lot cleaner. Maybe it is better to let this cool down a bit and see how things turn out in a few years.

MM: This is only an argument against the infix operator.

DD: Also, the example uses `function` instead of being a method in class.

KS: Unary Function Bind

```
::console.log  // -> console.log.bind(console)
```

AWB: This seems perfectly reasonable

MM: We could do the prefix notation without the infix notation

YK: What's the pedagogy here? When should programmers use `::`? MM: If you're calling a method, use dot. If you're extracting a method, use `::` YK: Why didn't we fix this with classes? MM: That's water under the bridge.

KS: Back to the infix operator. If the function is immediately called, then I'd like to be able to desugar to call:

```
obj::f();
// -> f.call(obj);
```

DD: What about with `new`?

KS: Don't do this desugaring with `new`

AWB: [back to the unary operator] I like this because it's an operation on a reference

MM: Can you write other expressions on the right-hand-side that evaluate to a reference?

AWB: DH's prior proposal allowed it

MM: The case I can think of is a variable reference, inside a `with`, that's resolved to an object property

AWB: No, that won't work in this case because it's not [...the right kind of reference?...]. The base is an object environment record in this case.

KS: Can this be a syntax error if it's not a property lookup?

DH: Why not require it to be a member reference?

MM: I don't see any reason not to statically constrain this to property lookup, either dot or square bracket

AWB: Some concerns, will consider offline

KS: [new slide: Bind, The Big Picture] the infix operator is an alternative to the adapter pattern or extension methods.

MM: `this` should really be passed as the first argument.

KS: The implicit `this` is really the first argument.

DH: Worried about the additional overhead this syntax adds to the language [referring to the infix operator]. Having to make the decision between `.` and `::` seems problematic.

JH: And in C# you always use `.`

YK: The alternative is you use an adapter first, and then `.` the rest of the way to the right

JH: Yes, you can do this with an adapter pattern, but then you need to build an adapter layer and decide how to do it.

BE: The infix operator seems attractive for extension methods

JH: I agree that it's sensitive to add new syntax, but this has been done in other languages and people adapt to it

DH: The way forward seems to be to get large-scale feedback about using this [maybe from Babel users?]

DD: React developers using Babel are already using the existing `::` implementation, and filed bugs when KS changed the semantics

BE: Agree with DH, need more data to see how users like it

EA: What AWB said earlier about the use of `this` rings true, ES6 just changed the uses of `this` (with classes and arrows)

[lots of discussion of other tokens: `->`, `.:`, `.?`]

Conclusion on bind

- Get more feedback from users of Babel

Part 2 of KS presentation: Private Fields

KS: Preface: this conflicts somewhat with decorators, in that it makes use of `@` for its syntax. Will focus on the semantics in this presentation.

KS: [slides]

JM: Is there something essential to private fields that makes it important that private fields are created at construct time and guaranteed to all be present on instances?

MM: It's important because you'd like to maintain invariants in your implementation, invariants you don't get anyway for public fields [due to them being public]

KS: Private fields should not write through the prototype chain [example from slides]

AWB: This just fails because the imposter object doesn't have the private field, nothing to do with prototypes

KS: Private fields should also not read through the prototype chain

AWB: It seems like that just falls out of the fact that the field isn't present on the receiver

JM: Trying to find common ground between private fields and initializers for instance properties

KS: Two main options around initialization: either initialize all private fields to `undefined`, then initialize them one by one, or run all initializers first, then create and write all fields at the same time. In the latter case, referring to `this` in the initializer expressions must be disallowed.

[lots of discussion about initialization, resulting in punting and moving on to discussion of how these things work after initialization]

KS: [presenting spec on github] https://github.com/zenparsing/es-private-fields

[discussion of the use of `PrivateMap` in the spec language]

MM: The PrivateMap is not reified in the spec text, which should avoid any complaints related to WeakMaps and the transposed representation

KS: Initialization: current spec text takes the "batching" approach discussed previously

AWB: Initialization of private slots involves walking through the inheritance hiearchy, gathering private field requirements at each level, and that gives us the necessary information for allocation. Then there's a staged initialization, where at each base class, the initializers end up running before super returns.

YK: Worried about return-override from super constructors. You have to add the private fields to whatever object is returned from super

ARB: Can we get away from the details of the initialization and step up to finish KS's presentation?

KS: Brand checks pass once initializers have run for a particular level of the inheritance hierarchy.

YK: Due to return override, you can't simply walk the inheritance hierarchy to gather the set of private fields.

AWB: It's important to me that the private fields are atomically allocated across the whole inheritance hierarchy.

BE: But you can't do that with return override.

BE: We could decide that return override is incompatible with these private fields.

BE: Or we could just figure out a way to let private fields be added each time super() returns

AWB: How do these things interact with proxies?

MM: You get a type error if you try to look up the the field on something that fails the brand check, and proxies fail the brand check.

MM: Another advantage to this proposal is that you can still add private fields to frozen objects.

--- MM branches off to talk about initializers, asking for them to initialize instead of assign, which would allow data-dependent const fields --

[discussion of possible constructor syntaxes for initialization]

YK: Worried that moving declaration of instance properties (private and public) into constructor would break decorators

MM: But then how do you decorate fields whose initialization is data-dependent?

YK: That's not a case that comes up very often.

DD strawman:

```
class Point {
  constructor(x, y) {
    private @x = x;
    public y = y;
  }

  get x() {
    return this.@x;
  }
}
```

March 26 2015 Meeting Notes

Brian Terlson (BT), Allen Wirfs-Brock (AWB), John Neumann (JN), Jeff Morrison (JM), Sebastian Markbage (SM), Yehuda Katz (YK), Dave Herman (DH), Alan Schmitt (AS), Lee Byron (LB), Domenic Denicola (DD), Kevin Smith (KS), Andreas Rossberg (ARB), Brendan Eich (BE), Erik Arvidsson (EA), Adam Klein (AK), Jordan Harband (JHD), Mark Miller (MM), Istvan Sebestyen (IS), Jafar Husain (JH), Rick Waldron (RW)

Private state continued

KS: the "nested stuff" part of the private state implementation might be more controversial, but please note that it's separable.

KS: Modified the V8 self hosted Promise to see what it would look like with "private state"

AWB: the most interesting use of private state is in the typed array hierarchy, especially in terms of subclassing

KS: presents https://github.com/zenparsing/es-private-fields/blob/master/examples/promise-before.js

KS: After: https://github.com/zenparsing/es-private-fields/blob/master/examples/promise-after.js

ARB: The implicit `this` in `@name` is not something we have done before in JS.

BE: CoffeeScript set the precedence. Are you suggesting that you always have to write `this.@name`?

AWB: The semantics does not change.

DD: The name is in a lexical scope

AWB: We could do the same with non private. That would not be a good idea.

JM: should there be symmetry between private and public properties?

BE: it's part of our job to make the ergonomics sweet and competitive.

KS: moving on to helper functions. If the helper function itself needs access to the private state, where can I put it? I can't put it outside the class anymore (like in promise-before.js), because then it doesn't have lexical access to the private state. So that seems to want to push that function declaration of the helper function into the class body. In the example here I'm mixing my proposals (with `this::_resolve(x)`), but you could have done it just as a function.

ARB: Might be cleaner to introduce @name methods. Both static and prototype methods.

JM: three alternatives here. What you have here, a lexically confined closure. Or, a static method that is "private". A third way is a private instance method (on the prototype).

KS: so you're talking about this kind of setup

```
class C {
  @x() {
    C.@x();
  }
  static @x() {

  }
}
```

I tried to go down this path. It seems weird that I have to do `this.constructor.@y()` (or C.@y()`)` `to call the`y`method from`x`.

AWB/ARB/others: no, that seems perfectly fine to me.

BE/AWB: The @x method is on the prototype.

MM: The visibility is that it is only visible inside the class body.

AK: I don't think `this.@x()` means the right thing, because we didn't want private field access to go through prototype chains.

(general acceptance that this is a deep issue)

MM: I now think they should be on the instance

DD: on the instance is bad ... memory-inefficient ...

BE: implementation doesn't have to work that way, even if semantics do

DD: I see ... so since they're private, we can lock them down enough that such an optimization is not observable

EA: These methods do not need to be properties anywhere.

MM: even in the weak map explanation, it's not like the closure pattern with separate-method-per-instance; it's a separate weak map per instance, but each of those weak maps points to the same function object.

(discussion of how this is related to vtables)

ARB: hold on, if this is private, why do you need a vtable at all... they're completely static

AWB: well ... what if you reify them ...

MM: reifying them is not what we're considering today; that'll be a much bigger fight.

ARB: it's basically a private scope.

MM: Compile time symbol table.

AWB: When you start to combine subclasses the tables makes things simpler because you can combine these tables.

ARB: not for methods ... for private methods it completely doesn't matter

BE: yes, it boils away. When we're talking about vtable it's purely about future things like interfaces

ARB: doesn't even matter for interfaces...

EA: what about call and apply? `this.@x.call(otherThis, ...)`

ARB: I think you can do that. And that's part of why I don't think we should allow a shorthand that omits `this.`

AWB: why would you ever do that?

DD: can you use super inside private methods?

MM: The home object for a `@m() {}` method needs to be the prototype (`C.prototype`) and similarly the constructor for a static at-method.

JM: Would it make more sense to think of these as functions that are set as instance properties during the instance property initialization.

MM: We want class private

ARB/MM: The --weak-- map explanation vs the private symbol is that the map case the function is associated with the object ....???

AWB: Or the vtable explanation.

EA: What if you had C.

```
class C {
  @a;

  m() {
    let somethingElse = C;  // <--
    somethingElse.@x();

  }
  @x() {
    C.@x();
  }
  static @x() {

  }
}
```

ARB: that is a very good point; I think it means we can't have the same name as both static and instance.

MM: It would force internal polymorphism; it's not good to pay the cost of this when it's essentially accidental.

ARB: given a random object `o` and you do `o.@x` on it, you have no way of knowing which type of private that is referencing statically (i.e. instance method private or static method private or instance data private), and that defeats the point.

MM/ARB: Do not allow them to have the same name.

DD: now this is bizarre. They act too different from public methods/public static methods.

EA: Kevin's original (nested declarations) idea is seeming more attractive after this exercise.

KS: (presents original nested declarations idea)

```
class C {
  @a;
  function xHelper(obj) {
    obj.@a;
  }

  m() {
    xHelper(this);
  }
}
```

(general admiration)

ARB: Weird about this is that it looks like a public declaration.

AWB: In this case you are not talking about dispatch.

ARB: at least it should not be function, maybe `private xHelper() { }`

KS: the concern is that if you use `private` then people would think it's a private method...

BE: `private function xHelper() { }` might help

DD: Not intuitive to programmers coming from other languages. People already use external helper functions.

KS: Feedback has not been too positive. The external helper function is used in Python for example.

ARB: fear that most programmers will want to use method syntax and will make helpers public just to get that

Immutable Records, Tuples, Maps and Sets

(Sebastian Markbage and Lee Byron)

https://github.com/sebmarkbage/ecmascript-immutable-data-structures

LB: ImmutableMap reutrns a new Map when you mutate them.

LB: Wants value semantics. Especially for == and ===.

MM: As well as `Object.is`.

BE: "NaNomali"

LB: Wants deep equality so that a Map of Maps or vector/records works too.

LB: Wants to work across realms and shared memory for workers.

LB: Based on Typed Objects.

Wants new syntax.

```
const xy = #{x: 1, y: 2};
const xyz = #{...xy, z: 3};

const xy = #[x, y];
const xyz = #[...xy, z];
```

LB: Record is the realm specific constructor/wrapper (like String)

BE: For value types we didn't want the wrappers.

JHD: What happens if you pass these primitives into the `Object` function? (like `Object('string')`)

LB: Works the same. It creates a wrapper of the correct type.

EA: Are the keys always sorted?

LB: Wants to allow implementations to do whatever what they want but...

MM: I think you are doing the right thing since then the comparison is cleaner.

AWB: Module namespace exports are sorted with the default comparison.

ARB: When you implement immutable maps you do not use hash tables but ordered trees.

MM: You cannot define an order for opaque object.

MM: Can an immutable map refernce a non mutable object?

ARB: You can specifiy that keys have to be deeply immutable.

LB: Would like to have same keys as in ordinary maps.

ARB: Then how would you implement this?

LB: You would use a hash function/method and a trie. The hash is optional. That is why we want it implementation dependent.

LB: Would have to define what order to use.

ARB: That means undeterministic behavior.

LB: In immutable.js the order is part of the equality.

MM: If you want you could canoninicalize the order before comparing.

LB: Most people do not depend on the order.

MM: You could do insertion order and make === be equal but make `Object.is` return false if the order is not the same.

ARB: Another option is to use value order where possible but use insertion order otherwise.

MM: You could use hash consing.

ARB: Does not help for ordering.

MM: Need to make sure that you do not use the real physical address, due to mutually suspicious code in the same realm. Need to be unguessable or use the hash to achieve information about code it should not know about.

YK: Maybe you want lower level primitives so that user code can implement this.

LB: No. We need to define the semantics so that VM can optimize this.

LB: A VM can use shared memory. It can use low level structs and memory to achieve.

ARB: A VM can have efficient per object hash codes.

JH: Also syntax.

AWB: Per object hash code is the big missing building block that we do not want to provide.

EA: It is not possible to make a user exposed hash code that is going to be as efficient as the internal hash code which might just be the address of the object.

MM: For ordering you might get away with a not very good pseudo random generator.

LB: It seems simpler and good enough to just stick with the insertion order for immutable maps.

MM: Implied cost. The precise equality would have to take the ordering into consideration.

LB: Batteries included philosophy.

AWB: You can still have a standard library that can be implemented in JS.

LB: Libraries can always provide more data structures.

AWB: I don't think we should provide new libraries without providing the primitives that allows these to be implemented in user code.

BE: What primitives are missing?

LB: Assuming we have typed objects and value types, value semantics

MM: Cryptographically safe pseudo-random numbers as a hash code.

DH: Sharing immutable objects across workers.

MM: Cryptographic pseudo random number generators.

MM: The number has to be large because collision is fatal

LB: Another issue is that we are creating a new value type for every record.

Conclusion/Resolution

- Identify the requirements.
- Progress report at a future meeting.

Composition Functions

(Jafar presents slides)

(Lots of discussion. Notes lost to a grue.)

Conclusion/Resolution

- Wide agreemeent that async functions are worth generalizing
- Wide agreement that promises are the dominant use case and should be the default
- Tricky problems with hoisting semantics, need more time to woodshed that problem
- async/await is wanted/needed so we should urgently figure out if this can ever be made to work

(discussion about wtf woodshed means - behind woodshed = to kill, to/inside woodshed = to spank)

Additional export __ from statements

(Lee Byron presents https://github.com/leebyron/ecmascript-more-export-from)

General agreement, suggestion to add Babel transpilation and fill out spec text, bring back at next committee meeting for a fast-track through stages.

64-bit math

(Brendan Eich presents https://gist.github.com/BrendanEich/4294d5c212a6d2254703)

Moves to stage