Minutes of the: ***Ad hoc video call of Ecma TC39***

held on: ***Wednesday the 7th of January 2015***

Attending

Dmitry Lomov, Andreas Rossberg, Allen Wirfs-Brock, Brendan Eich, Yehuda Katz, Brian Terlson, Dave Herman, Boris Zbarsky, Domenic Denicola, Mark Miller

Ad hoc hangout about subclassing

AWB: Gotta reach consensus soon, only a couple weeks to meet schedule.

AWB: general approach: allocation in base classes, [[Construct]] gets additional <original-constructor>

YK: and some cases of uninitialized `this`

AWB: that's another level, syntactically when can/can't you say `this`

YK: yeah that's been broad agreement for vast majority of thread; in nitty gritty details at this point. doubts that I have any agreement with those broad strokes?

AWB: worried that some of the stuff you said have danger of affecting deeper things, but if we can hold to [[Construct]] with \<original-constructor\> then we should be good

YK: only major disagreements are how many places where we don't agree should we have errors; couple places where we need some fine-tuning

YK: making more errors leaves us with more room to come to consensus later, so there's little to lose by making more errors

AWB: at spec level, it's a smell to have special cases and burden on impl

YK: question: I think in terms of special cases for `this` binding, we both have the same special cases, it's just when you hit the special case, in my semantics it's an error, in your semantics it's an allocation

DL: latest approach from allen doesn't have any allocation on it

YK: allen is saying he doesn't like errors because they add special cases

AWB: just want to minimize number of constructs with multiple possible errors depending on context

DH: Let's dive into the details - there are some error cases that will be conservative

AWB: let's talk about what we have consensus on. I think within this group there's consensus on overall approach

YK: lemme try to repeat it back:

- allocation happens in constructors
- new parameter to [[Construct]] that super uses to delegate original-constructor
- that parameter is used by base constructors to wire up prototype

AWB: yes

YK: I think that's the consensus

AWB: can we speak for tc39 on that approach and I can start revising the spec?

MM: before declaring anything, can somebody restate?

YK: thought I just did

AWB: I'll restate:

- allocation is in general pushed to base classes, merged with initialization into base class constructors

- enabled by adding original-constructor parameter to [[Construct]]

MM: does proxy trap take original-cosntructor?

AWB: yes

MM: do we have new.target syntax?

AWB: secondary issue, not part of current consensus; we're trying to decide what we do have consensus on. I'm guessing we don't on that yet, but I'm trying to identify what we do have consensus on that we can start moving forward on

MM: what about issue of how super is looked up?

YK: that's also not reached consensus at this immediate moment

AWB: part of core of this is super call in constructor is a [[Construct]] call up the constructor's prototype chain

YK: I do disagree with that, but there's a way to say that we'd all agree with: super call in constructor is a call to the superclass

AWB: I don't know what a superclass is

YK: that's the core issue, it's a source of disagreement at the moment but we don't have to address it right this second

MM: can I state precisely what we're agreeing on wrt super-binding issue:

- given static extends relationship between C1 and C2

- in absence of dynamic change of superclass relationship

- [[Construct]] trap in subclass that calls super calls [[Construct]] trap in superclass

YK: the only source of difference is whether there's a way to change it; yes we all agree in those cases

YK: nitty gritty details do matter and we should try to discuss

YK: allen, are you comfortable with the error cases laid out in your proposal?

AWB: I can live with them. I don't necessarily like them all. there is this issue about super in function-based constructors and essentially as my proposal lays it out, it's essentially illegal

YK: it's illegal today

AWB: right, according to those rules

YK: reason we arrived at that rule is there's a lack of agreement about whether functions containing super should do anything at all in absence of some reflective operation that tells them what they're doing

AWB: I don't want to talk about that now

YK: in light of that disagreement, we're stuck making it an error

BE: does anyone think we should not have the error?

DD: I think it's a shame it gets away from classes as sugar

BE: classes as sugar is a myth

YK: I agree classes as sugar is important, but <somebody fill in dherman will brb>

YK: The only source of disagreement about whether there should be a reflective capability to turn a regular function into a constructor. I don't think it's fair to say that this proposal diverges from classes as sugar.

DD: it's not a myth, you can do an ES6 desugaring that's correct

YK: super is already syntactically privileged.

DD: not with toMethod

YK: same deal with toConstructor

DD: my second point: we've been able to keep this equivalence with ctors using ES5 syntax. this proposal introduces the idea that ctors are something different. it's a shame that ctors in ES5 and ES6 no longer mean the same thing

YK: this is a rathole but I'm happy to do it

YK: ctors are not equivalent right now because you can't write super.foo()

AWB: you can

YK: you can't in a regular function-as-constructor

AWB: super.foo() breaks in all cases unless you toMethod

YK: right. so it's already the case that the equivalence is not entirely there

AWB: when you just see a function definition we don't know programmer's intent

YK: exactly right

AWB: doesn't matter in case of super.foo(). once a function has been made a method, and a constructor is a method, then you can say super.foo() and it has a well-defined semantics. independent of object creation / instantiation protocol that happens to use same keyword but is a completely different operation

DD: I need a clarification: I've been saying you could do it

AWB: gotta be careful: constructor is technically an instance method. it's the .constructor property of prototype, so you have to be careful to take function definition and wire it up to be class

DD: there's a temp function that gets created b/c toMethod needs to create copies

AWB: original function is no longer valid, so it's not useful to use a function declaration w/ a name. you'd use a function expression to get a temp function object

DD: I understand and appreciate the details. I get Yehuda's POV better now

BE: c-a-s is a useful goal but it's fine long-term, not needed immediately

YK: I think it's more useful to say it's a good goal but it's already needed

DD: I was meaning c-a-s for ES6 reflective operations

BE: which we don't have fully in hand

DD: you can desugar them as-is. now that I realize you have to create temp functions anyway, you can probably desugar yehuda's proposal

YK: once you have to call toMethod on ctor... I'm just arguing for toConstructor, not against c-a-s

DD: before this proposal we had classes as ES6 sugar, I'm not sure now

BE: we didn't have consensus on that prior situation

DD: but when we're evaluating proposals, one has classes as ES6 sugar, one does not

YK: I think we should care about it in the medium-term time frame

DD: I agree. this has been helpful, I think we can move on

AWB: DOM/WebIDL world has tendency to synthesize things that are not exactly ES6 abstractions but close

YK: agree that's important

BZ: to extend WebIDL. doesn't express ES6 classes we should fix it. may have sticking points but more we can converge the better

BE: there are degrees of freedom to converge in WC and WebIDL land

DD: I'm excited about coming discussions on webapps. it'll be interesting but I'm feeling optimistic

DD: ready to pop the stack

AWB: ok so we can declare consensus on that much

[Dave disappears briefly for family reasons]

YK: question is both proposals, there were some places where there is an uninitialized this. Main difference is, other than what cases there is uninitialized this, is what happens when you do. Original: allocation behavior. New: throw an error. Doesn't seem to add special cases.

AWB: We have to decide what we want.

YK: Yes, but I'm saying this isn't adding special cases.

AWB: Let me describe how it would be done if I controlled the world. Special cases means difference between functions that are created with a class or created with a function, etc.

[Damn, missed Yehuda's question :(]

AWB: Here's a case we have to deal with: two consecutive supers.

YK: I agree that doens't need to be a special case.

AWB: It's a case and we have to decide what happens.

YK: But if you describe this as a const binding it falls out of the semantics.

AWB: None of these are const bindings. This is not a TDZ. It can be modelled that way but in the spec it's not the same.

YK: I agree, but thinking of it as a const binding is coherent.

AWB: So you can say the second super is an error because super calls define `this` and therefore because `this` is already bound, the second is an error. It's an error before you make the call not after you make the call.

YK: In my world view, the this binding is just a const binding. I don't mean in the spec, I mean the model. All of my behavior falls out of that concept.

AWB: As long as we now make it clear that the super call is the initialization part of the const declaration for `this`.

YK: Yes.

MM: Problem. Assuming the proper way to write a subclass with an if with two supers. Then you statically have two const bindings. That's an error.

YK: Not static conditions, only semantic restrictions.

DD: Let's not model in terms of Const

MM: This is a useful analogy that's good for teaching.

AWB: The binding semantics for const work fine. It's the const decl static semantics that don't apply.

YK: It's unobsertvable if you don't change it

AWB: since you've never observed it...

YK: if you want to say it has one value, you have to say it has one value. idea it started off as value and no one checked... we're getting closer to uninitialized-this POV. may as well model it that way

MM: I find the TDZ on this attractive. what's the objection?

AWB: what do you mean?

MM: prior to super call, attempts to fetch its value is dynamic error

YK: way we've been modeling it, both me and Allen, is uninitialized this value. but question is what happens when you touch it

MM: can I try restating? this location is in uninitialized state

AWB: right

MM: attempts to fetch value while uninitialized results in dynamic error

AWB: yes. this isn't a variable binding. semantics of a reference to uninitialized this is what? we have several options. Yehuda is saying throw. there are alternatives

MM: I think you stated the two schools. the TDZ school has strong objections to lazy allocation. lazy allocation school: I don't understand the objections to TDZ

AWB: problem is that means constructors that don't contain super or in class declarations that don't have extend -- those have to reference this and they don't have a super in them

YK: that wasn't quite right. class declarations that don't have extend don't have to start world in TDZ state

AWB: I'm just talking about body of ctor.

MM: objection is TDZ semantics has to distinguish between subclass and base constructors, because TDZ is only for derived constructors, whereas uninitialized-this proposal doesn't

YK: the auto-allocation proposal

MM: sorry right. any other objection?

AWB: other than that I think they're equivalent

YK: there was a different objection, which is that if you don't have access to original constructor then child ctors that don't want to call super don't know how to wire up the prototype. but we worked through the fact that we all want that facility in the future

MM: does anybody here of lazy-allocation school object to the TDZ school strongly enough to block a declaration of consensus on TDZ?

MM: I hereby propose we declare consensus on TDZ

AWB: only thing I object to is calling it TDZ

MM: I'll call it whatever you want

YK: we have agreement

AWB: this has different initialization semantics in an extends constructor than a base constructor

MM: specifically, behavior it has in an extends constructor is it's born in an uninitialized state; attempts to read in that state throw; also have agreement that if base constructor is exited with this in uninitialized state with implicit return, then you have same dynamic error on implicit return?

AWB: this is where base class constructor differs from function

MM: yeah! you mean extended constructor

AWB: yes, right

DD: how crazy would it be to require base constructors to also call super

MM: what? doesn't make sense

AWB: they have a prototype

MM: that's a usability hazard

DD: we wanna teach people to always use super

AWB: it's specified that F.p is the function which is a constructor which creates an ordinary object

MM: I think of base classes and derived classes differently. even when they inherit from something. I don't think of them as being in a subclass relationship. I think of them as the root of a tree in a forest. in a derived class I know I'm thinking of something where I have to think about the base class. the derived class is defined in terms of a delta from a derived class. I do not find ti attractive to think of both base and derived classes as having a superclass

DD: ok nobody is enthusiastic about this remark, so withdrawn

AWB: lemme respond to Mark. I can't disagree but if you go down one level of thought, part of the issue here is that in classes there's class and instance side hierarchies. you might be talking about one side or other. for example if you have an abstract class and you have this base class. person who designed abstract class might've throw in constructor to tell people you can't instantiate this, so I wouldn't wanna do a super call. so when you go to next level of refinement, where these subtleties make difference. just black and white "I inherit" or "I don't inherit" says I always wanna do one way or another

YK: I agree with much of this. my experience is in Ruby. in Ruby they created mixin approach to deal with this issue from Smalltalk. I think we should explore this approach

AWB: Mark has to leave. I guess we need to decide where we're at and what the next step is

YK: I think only remaining source of disagreement, and I'm not sure how urgent, could wait till January since it's small impact on spec: is the semantics of super-lookup

AWB: worried about waiting till January

YK: I mean I think impact on spec is small regardless of which path we chose

DH: so let's keep discussing it on email and have another Hangout as necessary

AWB: also whether we expose new.target. in my proposal it's an optional step

DH: my feeling is I like it, I just don't want to introduce syntax this late in the game

AWB: I understand, but other side: if I put it in the spec now it's a one line delete if in a month it's a mistake. if it's not in in a couple weeks it'll be impossible to add for ES6

YK: I'm sympathetic to conservatism but ES7 is this weird parallel universe where things can land even sooner than ES6 stuff can

DD: I love new.target whatever the syntax. we can do it as fast as Array.prototype.includes! ;)

YK: that's my intuition. everybody agrees we need the feature. so let's make it a separate feature, commit to moving it as fast as we would if we felt the ES6 schedule pressure

DD: and yet allen's willing to spec it now...

YK: there's this weird opacity about ES6 process, so why not just do it as fast as we can in ES7 process

DL: my position in the mails: the semantic hole is big enough that people will write incorrect code and that's bad

DL: if this is consensus, that we can delay, that's fine. but I think we're close enough that we should just go and do it

DD: another perspective perhaps: as implementor you'd prefer to ship these features as a bundle regardless of when they're specced?

DL: no

DH: I don't think it's a common enough case to worry about incorrect code

YK: I think what Dmitry is saying is that return override is too buggy without new.target. My position is there are still many valid use cases for return override that don't require new.target. Even in derived classes, like the memoization and factory use cases. I don't think it's necessary to remove return override because there's even more obscure cases that need new.target

DD: can we make those function declarations?

YK: feel strange to remove just b/c some subset is hard to do w/o feature we're adding in future

DD: to be fair, when you declare something to be a class there's a weak implicit contract that objects you return should derive from correct prototype

YK: memoization case is one where you're fine

BT: I understand Dmitry's use cases and how people can get into trouble, but I don't think that most devs will ever come across that kind of code so I'm not sure that's that big of a risk

DD: since we're gonna ship new.target syntax quickly anyway, let's put return override in same package

AWB: return override is there. it's more work to remove it

YK: returning anything from derived constructor would become an error. it seems like more work to remove it

AWB: I'd be reluctant to remove it. people would start complaining we've broken the language

DH: proposed plan of action: kick the can down the road by a couple weeks; let's discuss the syntax on es-discuss, try to drive it to a conclusion and then revisit the decision of exactly what ships when maybe in January f2f, maybe Feb

[broad agreement]

AWB: I'll post this proposal

DD: I suggest a repo called "scratchwork" instead of a gist, so you have revision history

AWB: we agree it's essentially as outlined in proposal?

YK: I think the only point that's not fully agreed is the point Dmitry had you add

AWB: that was always implicit in it. we don't wanna repurpose HomeObject to have broader meaning than it currently has

YK: we're out of time here. we actually don't agree

AWB: the spec has to do something. the spec can't say somethign we figure out what the superclass is

YK: I agree spec has to do something. that doesn't mean we have to say what spec has to say in the proposal. it's a detail we're still nailing down.

AWB: I have to put something in the spec.

YK: Allen I think you should keep doing what you've been doing with the spec, but I am not comfortable ending that conversation. AFAICT Brendan and Dave have similar concerns based on their emails, and Mark also expressed concerns. I think we should keep having the discussion

AWB: to be clear, you're proposing changing what it means to set prototype of a function

YK: what I'm proposing is a new slot on constructor which is the home of the naked super call

AWB: remember constructors are just functions

YK: yep, but it's an internal slot and main point is it does not allow mutation of that particular super

AWB: that means you can't do __proto__ wiring of prototype of constructor in a meaningful way

YK: today if you do __rpoto__ wiring your static and instance methods still point to original place

AWB: where static methods point is independent of what the [[Construct]] supercall does

YK: exaftly. it's already the case that if you do the wiring, several attempts to look up superclass point at original static

YK: the issue is whether two uses of name super can have different meanings, one static, one dynamic

YK: in one case, find my superclass is static and find my superclass is dynamic

DH: the issue is what Yehuda's gist points out

DL: the gist is wrong

DL: so super.m() is [[HomeObject]]._proto_.m()

note [[HomeObject]].m()

so this is still dynamic lookup in both cases

YK: the point is super means in user's head "get my superclass" but that means different things in different contexts

AWB: that's not what it means at all. it is "continue to lookup at the point where this method was found"

YK: I agree those are the semantics

AWB: that's what the semantics are supposed to be. HomeObject is a good enough approximation for all cases

YK: what I'm suggesting is that if HomeObject is a good enough approximation for methods, then HomeObject or some other similar approximation is good enough for constructors

AWB: super was exactly equivalent originally to calling super.constructor. an instance-side lookup

YK: agreed

AWB: bad effect was if somebody changed value of .constructor on instance side then who knows where that call goes to

YK: agree that was bad

AWB: constructor side is constructor side operation. shouldn't be on instance side

YK: agree

AWB: who's next up that constructor's prototype chain

YK: more precise analogy: expect super() to talk about same thing as super property lookups in static method

AWB: no because nothing static about static method. it's just a dynamic method on a constructor

YK: exactly. so when you call super.foo() on static method you're talking about superclass constructor

AWB: no. you're talking about where this method was found. could be intermediate place in heirarchy

YK: I fully understand the semantics. what I'm saying is in both static method and constructor there's a conceptual step which is "I need my superclass constructor right now". I would like those to return the same thing even though the operation is different

AWB: I don't see it that way

AWB: a super.foo method call is a super.foo method call. doesn't really matter what sort of method it's in. wherever that shows up.

YK: I agree

BE: guys, we have to end this meeting soon and you're rehashing. Yehuda's gist AIUI, even with nits uncorrected, shows a divergence between constructor and method

AWB: they're not the same thing!

BE: I know. that's an open issue. that's the point we're discussing. we're not gonna solve it by saying that's the way it is, or the way it shouldn't be, we need to say why

YK: only way to make progress is for people beyond me and Allen to speak on this

DH: we just don't have consensus on this point in Allen's proposal. the whole rest of the proposal has consensus

YK: 7.5/8

BE: 7.5/8 is pretty good :)