Minutes for the: *32nd meeting of Ecma TC39*
in: *San Francisco CA, USA*
on: *29-31 January 2013*

1 Opening, welcome and roll call

1.1 Opening of the meeting (Mr. Neumann)

Mr. Neumann has welcomed the delegates.

Companies in attendance:

Mozilla, Google, Microsoft, eBay, jQuery, Yahoo, Northeastern University, UCSC

1.2 Introduction of attendees

John Neumann (MS, Mozilla, Yahoo, Google)

Norbert Lindenberg - Mozilla

Nebojsa Ciric - Google

Allen Wirfs-Brock - Mozilla

Luke Hoban - Microsoft

Brian Terlson - Microsoft

Sam Tobin-Hochstadt – Northeastern University

Andreas Rossberg - Google

Erik Arvidsson - Google

Yehuda Katz - jQuery

Rick Waldron - jQuery (took minutes)

Eric Ferraiuolo - Yahoo

Matt Sweeney - Yahoo

Douglas Crockford - eBay

Brendan Eich - Mozilla

Waldemar Horwat – Google

Tim Disney – UCSC

Daily participation:

 # January 29 2013
John Neumann (JN), Norbert Lindenberg (NL), Allen Wirfs-Brock (AWB), Rick Waldron (RW), Waldemar Horwat (WH), Tim Disney (TD), Eric Ferraiuolo (EF), Sam Tobin-Hochstadt (STH), Erik Arvidsson (EA), Brian Terlson (BT), Luke Hoban (LH), Matt Sweeney (MS), Doug Crockford (DC), Yehuda Katz (YK), Nebojša Ćirić (NC), Brendan Eich (BE)

 # January 30 2013
John Neumann (JN), Norbert Lindenberg (NL), Allen Wirfs-Brock (AWB), Rick Waldron (RW), Waldemar Horwat (WH), Tim Disney (TD), Eric Ferraiuolo (EF), Sam Tobin-Hochstadt (STH), Erik Arvidsson (EA), Brian Terlson (BT), Luke Hoban (LH), Matt Sweeney (MS), Doug Crockford (DC), Yehuda Katz (YK), Nebojša Ćirić (NC), Brendan Eich (BE), Andreas Rossberg (ARB)

 # January 31 2013
John Neumann (JN), Allen Wirfs-Brock (AWB), Rick Waldron (RW), Waldemar Horwat (WH), Eric Ferraiuolo (EF), Erik Arvidsson (EA), Luke Hoban (LH), Matt Sweeney (MS), Doug Crockford (DC), Yehuda Katz (YK), Andreas Rossberg (ARB), Sam Tobin-Hochstadt (STH), Brendan Eich (BE),

1.3 Host facilities, local logistics

On behalf of Mozilla **Brendan Eich** welcomed the delegates.

1.4 List of Ecma documents considered

Ecma/TC39/2012/086	Minutes of the 31st meeting of TC39, San Jose, November 2012
Ecma/TC39/2012/087	Draft minutes of the conference call of the IPR adhoc on 3 December 2012
Ecma/TC39/2012/088	Eighth draft Standard ECMA-262 6th edition, December 2012
Ecma/TC39/2013/001	Venue for the 32nd meeting of TC39, San Francisco, January 2013
Ecma/TC39/2013/002	International registration of the "ECMAScript" trademark in Japan
Ecma/TC39/2013/003	Draft TC39 RFTG Scope
Ecma/TC39/2013/004	Agenda for the 32nd meeting of TC39, San Francisco, January 2013 (Rev. 1)
Ecma/TC39/2013/005	Submission of ECMA- ECMAScript to the European Commission
Ecma/TC39/2013/006	Draft minutes of the conference call of the IPR adhoc on 15 January 2013
Ecma/TC39/2013/007	Current status of files for the TC39 Royalty Free patent policy
Ecma/TC39/2013/008	Draft minutes of the conference call of the IPR adhoc on 4 February 2013

2 Adoption of the agenda (2013/004-Rev)

The agenda was approved.

3 Approval of minutes from November 2012 (2012/086)

The minutes of the September 2012 TC39 meeting have been approved as presented. Individuals that took technical notes were recognized and appreciation extended. **Rick Waldron** volunteered to take technical notes. The technical notes – which reflect the discussions correctly and has been already shared within TC39 - are included in Annex 1 of the minutes.

4 Discussion of ES harmony (technical contributions are available and can be found on the ES wiki)

4.1 "is/isnt" operators

4.2 Is SameValue really what we want for Map/Set equivalence? (the -0 different from +0 issue)

4.3 Parameterize the equivalence operator for Map/Set?

4.4 Is there a need for Number.isNaN if we have Object.is available?

4.5 Why standardizing on __proto__ and not __define(G|S)etter__, __lookup(G|S)etter__ ?

In a recent es-discuss thread, Brendan stated:

[...] So the full answer to Andreas and others' query about why __proto__ normative mandatory in ES6 but no __define/lookup... is:

1. __proto__ much more used on the mobile (iOS WebKit-first) web, no equivalent interop pressure for __d/l.

2. ES5 is in all new/evergreened browsers and it has standard APIs supplanting __d/l but nothing for writing to __proto__.

Therefore __proto__ gets standardized, __d/l do not.

Offlist, I mentioned that we should add this discussion to the agenda for next meeting so that we can say this out loud, entered as a resolution in consensus, and then published as a matter of public record.

1. Rationale for not adding Object.setPrototypeOf

https://mail.mozilla.org/pipermail/es-discuss/2012-May/022904.html

Brendan

4.6 A. Kevin Smith's grand compromise proposal for ES6 strict and sloppy semantics

(https://mail.mozilla.org/pipermail/es-discuss/2012-December/027736.html):
> (1) No opt-in required for new syntax, except:
> (2) No breaking changes to sloppy mode, and
> (3) No grammar contortions (e.g. let) to support sloppy mode. And
> (4) All new syntax forms with code bodies are implicit strict.

Item (2) raises the question of what to do about function-in-block,
separated out here as item C below.

B. Should sloppy-mode function-in-block usage on the web be broken, with specific evangelism of 20 or so sites relying on its de-facto intersection semantics that don't intersect with ES6's?

See
https://mail.mozilla.org/pipermail/es-discuss/2012-December/027419.html
et seq.

4.7 Fail-fast destructuring with ?-syntax for irrefutable opt-in:

>/1. No ToObject(RHS).
/>/2. Exception on missing property selected without a ?-suffix.
/>/3. ?-suffix allowed on any pattern, imputing undefined deeply instead
of refuting. /
>/4: the ? is a separate lexeme from the : in long-hand patterns.
/
One entry point:
https://mail.mozilla.org/pipermail/es-discuss/2013-January/027800.html,
wherein Allen argues cogently against (1) but endorses (2-4).

Brendan

4.8 Refactored new operator and the @@create method

4.9 Update on built-in subclassing based upon @@create

4.10 data to share on real world javascript usage (function-in-block, duplicate parameter names, const, strict mode, var let/let[x]=1, and maybe more) if people are interested. Could also just distribute the deck if the agenda is tight?

4.11 Name property of functions

http://wiki.ecmascript.org/doku.php?id=strawman:name_property_of_functions

4.12 TypedArrays: specification progress and open issues (Allen)

4.13 Revisit the @-names discussion/resolution from the November meeting

A significant body of work has emerged that I feel makes a strong case towards making an exception to the cut-off deadline. I'd like to urge everyone to take a moment to review this: https://github.com/Benvie/continuum/tree/gh-pages/engine/builtins

5 Edition 5.1 Issues

No discussion.

6 Second edition of ECMA-402

6.1 Status report

A good summary of the discussions please find on page 11 and 12 of this report.

7 Test 262 Progression

7.1 Status report

No discussion.

8 Status Reports

8.1 Report from Geneva

8.1.1 Brief report from the IPR meeting

Mr. Neumann gave a brief report on the work of the IPR Adhoc Group. The group has not finished its work yet.

Up to now the Adhoc had 14 meeting, the minutes have been distributed also to TC39 for information. Any comments and feedback should go to the IPR Ad-hoc directly or to the Ecma GA. In TC39 there are no plans to discuss these issues.

The Ad-Hoc gave the “homework” to **Mr. Neumann** and **Mr. Wirfs-Brock** to get an update of the scope of TC39 and to prepare a draft scope and Programme of Work for the TC39 RF TG. This was completed and the IPR Adhoc has approved it (TC39/2013/003/Rev1). Before further progressing by the Ecma General Assembly the IPR Adhoc has asked TC39 to endorse the content of the document.

8.1.2 Endorsement of the TC39 Royalty Free Task Group scope

Mr. Neumann introduced the updated scope and Programme of Work for TC39:

TC39 RFTG SCOPE:

Standardization of the general purpose, cross platform, vendor-neutral programming language ECMAScript. This includes the language syntax, semantics, and libraries and complementary technologies that support the language.

Programme of Work:

1. *To maintain and update the standard for the ECMAScript programming language (such as all editions of ECMA-262, ECMA-357).*

2. *To identify, develop and maintain standards for libraries that extend the capabilities of ECMAScript (such as all editions of ECMA-402).*
3. *To develop test suites that may be used to verify the correct implementation of these standards (such as ECMA TR/104).*
4. *To contribute selected standards to ISO/IEC JTC 1*

Note: This is the version that got support from the IPR Adhoc in their meeting of January 15, 2013

TC39 has unanimously endorsed the scope and programme of work of the possible future TC39 RF TG.

Mr. Neumann has requested the delegates again to study with their legal experts the documents distributed in TC39/2013/007, and if there are comments, questions they should be directly addressed to the CC or the Ecma Secretariat.

9 Date and place of the next meeting(s)

Schedule 2013 meetings:

- March 12 – 14, 2013 (Yahoo, CA)
- May 21 – 23, 2013 (Google, United Kingdom)
- July 23 – 25, 2013 (Redmond, WA)
- September 24 – 26, 2013 (Notheastern University, Boston, MA)
- November 19 – 21, 2013 (PayPal, CA)

10 Closure

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

Annex 1

Technical Notes (by Rick Waldron):

 # January 29 2013 Meeting Notes

John Neumann (JN), Norbert Lindenberg (NL), Allen Wirfs-Brock (AWB), Rick Waldron (RW), Waldemar Horwat (WH), Tim Disney (TD), Eric Ferraiuolo (EF), Sam Tobin-Hochstadt (STH), Erik Arvidsson (EA), Brian Terlson (BT), Luke Hoban (LH), Matt Sweeney (MS), Doug Crockford (DC), Yehuda Katz (YK), Nebojša Ćirić (NC), Brendan Eich (BE)

JN: …Welcome and introduction, agenda. Discussion of TG/TC procedural changes. Please review and feedback.

…Adoption of the Agenda.

WH: All of the initial agenda items are missing references. Can't figure out what some of them are referring to from just their titles.
WH, JN: Please put proposal links (or other references) into agenda items like we did in the past.

AWB: Are there items that need to be discussed that aren't yet on the agenda?

STH: Will provide an update on Modules

Added: 4.14 Modules Update

AWB: Discussion about Proxy?

STH: A lot of discussion about the interaction of Proxy and Private Names

…Should we wait for Mark Miller?

AWB: Nowhere near consensus on any point of discussion from the mailing list.

Added: 4.15 Proxy Issues

LH: (to AWB) did you want to give an update on the spec?

Added: 4.16 Spec Status Update

JN: We should have a discussion regarding the list of items that will actually end up in ES6 and determine exact ES6 additions.

RW: Let's set this as an agenda item for the next meeting and try to reduce the number of submitted agenda items. This allows everyone the opportunity to prepare for the large scale discussion.

JN: Will add an agenda item for March.

…Brief discussion about features that may or may not be ready; STH identifies: event loop (dependency of modules, Object.observe)

AWB: Two issues: Semantic completeness of the designs and what does it take to implement these into the language.

WH: Are proxies ready?
AWB: Not yet.

Mixed discussion.

JN summarizes as part of 4.16 discussion

Agenda Approved

November 2012 Minutes Approved

4.16 Spec Status Update
(Presented by Allen Wirfs-Brock)

AWB: Revisits Nov 22, http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts

Discussion re: instanceof.

EA: Checks and changes to implementations that use instanceof

STH: Expect slow down from use of instanceof?

AWB/EA: yes.

WH: Concerns, how do you tell if a regex is a regex

YH: Not with instanceof, use Object.prototype.toString

AWB: Shouldn't use instanceof

EA: This hook makes instanceof behave they way you expect it.

AWB: Revisits Dec 21, 2012 Draft changes

Mixed discussion re: keeping up to date with the spec drafts, request for more technical-focused reviews.

__proto__.

LH: Begins with raised points about __proto__…

Mixed discussion about __proto__ semantics.

AWB: There is an underlying feeling that we don't like this and we need to get over that.

LH/YK: Agree.

DC, WH: Disagree.

YK, DC: Discussion about __proto__ reality.

YK: We need consistency.

DC: We should not standardize and leave as is.

Impasse.

LH: Similar issues with block scoped function declaration incompatibilities.

WH: We should give up on __proto__ in the same way that we gave up on "with"

EA: IE is implementing for web compatibility

AWB: The defacto standard for mobile web uses __proto__

EA: IF we could go back, we would do it differently

YK: We just need compatibility

LH: We need to just suck it up and standardize

Mixed discussion re: user code and __proto__

AWB: (clarifies) Luke is requesting that we pin down the __proto__ details as soon as possible.

WH/DC: (have concerns, against standardizing __proto__)

YK/BE: Discussion re: interop with current implementations.

BE: (Review of latest changes to __proto__ in Firefox)

EA: Matches Safari

BE: __proto__ is configurable (can be deleted), accessor (getter and setter throw), reflection does not leak.

AWB: Involves magic

BE: Yes, but minimal. (Confirms that latest __proto__ is out in wild, Firefox)

WH: Clarify "poisoning"?

BE: When you call it, it throws.

WH: So how does it know when not to throw? (If it always throws then it won't work.)

EA: Throws if called with object and setter coming from different realms

…Discussion re: MOP semantics with __proto__

BE: Proxy has to stratify the MOP.

AWB: Another issue… Objects that are non-extensible, can you change __proto__? Specifically, now that we're talking about being able to change __proto__, what type of objects can be changed?

BE: Wait for Mark?

YK?: Changing __proto__ is a write, not adding a property, so it should not be affected by extensibility.

AWB: Agree.

Hold until Mark Miller is present.

EA: ES5 requires [[Prototype]] to be read only when the object is non extensible.?

(Further discussion)

BE: Let's get back to this with Mark present and come to a conclusion.

4.1 is/isnt operators
(Presented by Allen Wirfs-Brock)
http://wiki.ecmascript.org/doku.php?id=harmony:egal

AWB: The plan of record on the wiki is that there is an "is" and "isnt" operator. There are various discussions on the mailing list that refer to this, but also to Object.is and Object.isnt. We need a _final_ decision on the operator form.

YK: The discussion was to move the world from ==?

YK: The differences between === and is are significantly small.

BE: On behalf of Dave Herman and Jason Orendorff… Syntax needs to be worth the expense and there options.

General agreement.

WH: (question about new type NaNs)

AWB: No spec provisions, currently…

WH: Hypothetical discussion about new "decimal" type with a "decimal NaN" and the implications of NaN equality. The issue that would arise would be that a decimal NaN would be unequal to itself but would be distinguishable from the regular Number NaN. Therefore the example code for how *users* could implement `is` wouldn't work. However, if `is` is a language feature, then an implentation can fix it up to behave correctly when it adds additional primitive types such as Decimal, just as it would fix up the behavior of ==, etc.

Conclusion/Resolution
Consensus on **No operator**

Do we want API? **Yes.**

What is the name? **Object.is(x, y)**
(NOT Object.isnt, ! is sufficient for negation)

If it's "all discriminating", should it discriminate +0/-0? **Yes**

NaN is _not_ different from NaN (all NaNs are equated).

Every observable value "is" itself and nothing else.

4.4 Is there are need for Number.isNaN if we have Object.is available?

BE: Keep the concerns separate.

AWB: If we have Object.is, we don't actually need Number.isNaN

DC: Object.is is sufficient.

BE: (Whiteboard example)

RW: Punt on Number.isNaN for ES6, defer to ES7 hinging on need based on potential emergent library code.

WH: Number.isNaN is analogous to other Number functions such as Number.isFinite and Number.isInteger. It should stay.

(toInteger was mistakenly changed to toInt, discussion leads to revisiting March 29th 2012 notes where no rationale was recorded. https://mail.mozilla.org/pipermail/es-discuss/2012-March/021919.html)

Given the existence of the other Number.is* functions, several people flip their views.

Conclusion/Resolution
Overturn the Number.toInteger=>toInt change. Restore as toInteger. (Brendan, can you fill in the rationale for this?)

Number.isNaN remains in ES6

4.2 SameValue for Map/Set equivalence?
4.3 Parameterize the equivalence operator for Map/Set?

AWB: For ES6, only specify a limited set of comparison operators?

Discussion re: arbitrary comparisons?

Hash Codes?

STH, WH, several others: Need to support strict (`is`) equality at a minimum.

YK: If the goal is to avoid whole new design issues, stick to strict equality.

BE: Reminder that adding hash codes now is too late in the game.

YK/AWB: Concerns that maps with strict equality will trip people up.

YK: Typically, Maps have custom equality capabilities.

AWB: (Reminder of hash code pitfalls) But _only_ strict equality will be a disaster, but can't introduce hash codes at this stage.

AWB: [if we support both strict and other kinds of equality] suggest the default: +0/-0 are the same; NaN is a value, that is equal to itself for Map/Set equivalence.

Assuming we have a mechanism to override…

Ideally, we'd want to set this in the constructor, but there is already a single optional initializer argument.

BE: Let's find a champion to work this out over the next two days.

AWB: I can do this.

Conclusion/Resolution
To revisit, pending proposal from AWB. (See Jan 31 notes)

Must address the minimal requirements on the wiki

6 Second Edition of ECMA-402 (Internationalization)
Last Update: https://github.com/rwldrn/tc39-notes/blob/master/es6/2012-11/nov-27.md#internationalization-update

NC: Approved to continue work. APIs covered, people assigned to write strawman. Have API changes to be written into the spec, based on use cases that have emerged.

NL: Normalization in the language specification: talked about 2 additional normalization forms.
1. CF, Case folding
2. NFKC_CF, Normalization form, Compatibility Decomposition,
followed by Canonical Composition, combined with case folding

NL: Chrome 24 shipped with the first implementation of Edition 1, not prefixed any longer

NC: Overrode the old methods String.prototype.localeCompare, Date/Number.prototype.toLocaleString

BE: Good so far?

NC: Have some bugs with performance, but can address.

EA: No issues with actual semantics?

NC: No complaints.

JN: Schedule? On track?

NC/NL: So far, but rely on ES6.

Discussion about overlapping schedule. Current plan for Intl 2 is for approval in June 2014, but ES6 may slip to the same date. NL wondering whether TC 39 can review two specs at the same time; AWB thinks that should be OK.

Notes from NL/NC:

December 12: Ecma GA approves ECMA-402
http://ecma-international.org/publications/standards/Ecma-402.htm

January 10: Google shipped first implementation without prefix in Chrome 24

December 14: Internationalization ad-hoc met

- discussed scope in more detail, assigned strawman writing

Since then: Strawmen written, but not reviewed yet:

- number parsing
http://wiki.ecmascript.org/doku.php?id=globalization:numberparsing

- text segmentation
http://wiki.ecmascript.org/doku.php?id=globalization:text_segmentation

- message formatting
http://wiki.ecmascript.org/doku.php?id=globalization:messageformatting

Strawmen assigned, but not written yet:

- character properties (regex or API?)
- display names
- time duration formatting
- alphabetic index

API changes to be written up:

- normalization (language spec - add CF and NFKC_CF)
- case conversion
- time zone support
- script reordering in Collator
- pseudo-numbering systems in NumberFormat

4.5 Why standardizing on __proto__ and not __define(G|S)etter__, __lookup(G|S)etter__?
(Based on a recent es-discuss thread)

Why __proto__ normative mandatory in ES6 but no __define/lookup... is:
1. __proto__ much more used on the mobile (iOS WebKit-first) web, no equivalent interop pressure for __d/l.
2. ES5 is in all new/evergreen browsers and it has standard APIs supplanting __d/l but nothing for writing to __proto__.
Therefore __proto__ gets standardized, __d/l do not.

Rationale for not adding Object.setPrototypeOf
https://mail.mozilla.org/pipermail/es-discuss/2012-May/022904.html

Conclusion/Resolution
Consensus that __define(G|S)etter__, __lookup(G|S)etter__ will not be standardized, nor added to an Appendix.

4.8 Refactored new operator and the @@create method
(Presented by Allen Wirfs-Brock)

Slides (pdf):
http://wiki.ecmascript.org/lib/exe/fetch.php?id=meetings%3Ameeting_jan_29_2013&cache=cache&media=meetings:subclassing_builtins.pdf

The Basic Issue:
* Object Allocation and object initialization are separable issues.
* Subclassable abstractions require program level control…

Why Doesn't this work?

```js
class Vector extends Array {
  constructor(…args) {
    super(…args);
  }
```

```
}

let a = new Array();
let v = new Vector();

a[5] = v[5] = 5;

a.length; // 6
v.length; // 0
```

Because…

* Array uses a special exotic object representation that changes the semancs of [[DefineOwnProperty]].

* The object that new Vector creates and passes to the Vector constructor is an ordinary object, not an exotic Array object.

* Even with the super call, the Array constructor doesn't transform its this object into an exotic Array

First Solution

* Use normal [[Construct]] for built--ins.
  - Move magic ini1aliza1on (internal data proper1es and internal methods) into the constructor func1on, post object alloca1on.
* Internal data proper1es need to be expandos (probably based upon private symbols)
* Built--in methods use internal data property sniffing instead of [[Class]] brand check.

Objections

Ollie's Objections:
* Doesn't want internal data properties to be expandos
* Implementors want to allocate different machine/C level data structures for different kinds of built-ins.

Allen's Objections:
* What about internal method conflicts?

WH: Jason's Objections:
* More than one magic constructor can be applied to an object

```js
let d = new Date();
Map.call(d);
console.log(d.getYear()); // 2013
Map.prototype.set.call(d,"month", "January");
console.log(Map.prototype.get.call(d,"Month")) // January
```

What do other dynamic OO languages do?

* Separate object allocation and initialization into separate phases.
* The "shape" and special characteristics of an object are fixed during the object allocation phase.
  – Kind of like what [[Construct]] does, but...
* The allocation phase is defined as a separate class method
  – Can be inherited, or over-ridden, or super-invoked by subclasses.

Sounds Good, Let's see if it works for JavaScript

* @@create is a well known symbol that when used as a property of a constructor provides the alloca1on method for that constructor.
* New definition of the ordinary [[Construct]] :
1. Let creator be [[Get]] of this constructors @@create property
2. If creator is not undefined, then
a. Let obj be the result of calling creator with this constructor as its this value.
3. Else, fallback implementation
a. Let obj be a new ordinary object with its [[Prototype]] initialized from this constructor's "prototype" property.
4. Call this constructor with obj as the this value and the original argument list.
5. Return obj.

BE, EA, YK, WH, others: Get rid of the fallback implementation and just throw if there is no @@create. It's unnecessary complexity.

* Most constructors just inherit Function.prototype[@@create] which allocates a new ordinary object with its [[Prototype]] initialized from this.prototype (the constructor's "prototype" property).
* `new Foo() <=> Foo.call(Foo[@@create]())`

TODO: Copy slide text to notes?

Discussion re: `class Foo extends ??` semantics discussed in July, https://github.com/rwldrn/tc39-notes/blob/master/es6/2012-07/july-26.md#maxmin-class-semantics

WH: What happens when you do Number[@@create]?

AWB/EA: Creates a new wrapped Number object.

BE: `new Number()` <=> `Number.call(Number[@@create]())`

AWB: Built-in @@create methods are non-writable, non-configurable. Just like built-in "prototype" properties (From slides).

… Back to slides.

@@create Also Useful for Application Classes

* DIY Branding
```js
import $$create ...;
const $fooBrand = Symbol(true); // or use a WeakMap
export class Foo {
  isFoo() {
    return !!this[$fooBrand];
  }
}

Object.mixin( Foo, {
  [$$create]() {
    let obj = super();
    Object.defineProperty( obj, $fooBrand, {
```

```
      value: true
    });
    return obj;
  }
});
```

(note: `$$` sigil for illustration purpose only)

WK: The @@create could be hijacked and used to brand some other object with my object's brand.

AWB: No different from hijacking from an Array or Date…

WK: Good for branding case, not for security

AWB: (Gives example of how you might make it work). Explanation of creating Proxy based instances with @@create (see slides)

ES5 Built-In Branding

Consider ES5+ Reality:

```js
function T() {
  var obj = [];
  obj.__proto__ = T.prototype;
  return obj;
};

var t = new T();
t[5] = 5;

t.length; // 6
Array.isArray(t); // true
t.toString(); // [object Array]
```

Built-in branding is based on the shape and capabilities of the actual instance object.

ES6: @@create Determines Branding

* Array.isArray will report true for subclass instances that are built-in exotic Array objects. These are allocated using Array[@@create];

```js
let v = new class extends Array {};
Array.isArray(v); // true
```

* Unless over-ridden using the `@@toStringTag`, `{}.toString` will report the legacy [[Class]] for built-in subclass instances if they are allocated using a built-in @@create method.

WH: If you're call isArray and it returns true, you'd expect to call Array.prototype methods.

AWB: No guarantees about the prototype chain

MM: Even if Array is frozen, the Array.prototype is mutable and I can add my own malicious push, pop, join.

MM, AWB, WH: (Discussion of security concerns)

Agreement that none of this weakens any existing invariants.

No Need for [[Class]] or [[BuiltinBrand]]

* These are really just specification devices for talking about specific forms of objects.
* Spec has always also used language like "an Array object" or "a RegExp instance"
* In ES6 spec. all [[Class]] uses uses can replaced with language like:
    - "is an exotic array object" <=> [[Class]]=="Array"
    - "has a [[Match]] internal data property" <=> [[Class]]=="RegExp"

File:
[[Match]] => [[RegExpMatch]]

Testing the |this| value almost works

(See slides for code example)

Conclusion/Resolution
Consensus on @@create, allocation and initialization decoupling.

Constructors need to be able to recognize initialized instances

* Built-ins can do this via existing internal properties
* User code could roll its own

Probably better to formalize initialized state as part of ES object model

* Add one bit of state to every object: initialized/unitialized.
* Built-in @@create methods set new object state to uninitialized.
* `Object.call(uninitObj)` and other built-in constructors set initialized this objects to initialized state.
* `Object.isInitialized(obj)` is a new method that only returns false if obj is an object that is in the initialized state.
* Object.create(), {}, [] and various built-in functions create objects that are already initialized (backwards compat)

Semantics…

```js
// Decoupling allocation and initialization:
// Calling new X()…
class X {        // 1. X[@@create](), allocation
  constructor() { // 2. Initialization
    super();
    this.state = "ready";
```

```
  }
}
```

EA: Worth the additional bit? Is the need common enough to address with new language API surface?

What about @call?

MM: @construct? If one is specified, otherwise … super call vs. super construct (Mark—Sorry, I had trouble following this—can you fill this in? Thanks!)

MM: Are there any objections to the parallel @construct entry point.

YK: It makes less sense in the ES6 context, since classes are focused around defining a constructor…

MM: I see, then call is the odd-ball. Agree with the objection.

YK: @call makes sense in the class case, @construct makes sense in the function case.

AWB/YK: Need statics to make the @call

Mixed discussion regarding "static"

Yehuda will draft a static strawman for tomorrow.

(Break)

YK: All functions have a call and construct property, I propose to expose them

AWB: All functions are constructible, when a function is "newer", something happens before the function is invoked.

AWB/YK: (Discussion re: existing semantics of function call and construct)

BE: The idea of having state in objects for initialization…

WH: The main point of the discussion was that many are uncomfortable with adding an extra bit of state to objects to define whether they're initialized or not and, worse, that state is not a reliable indicator of whether a function was called via Foo() or new Foo(). Someone could forget to set the initialized bit (in a variety of ways) and disrupt further code at a low level. The consensus seemed to be to get rid of that state and instead use either separate call and construct code or some way to detect whether code was called as a function or as a constructor, although we hadn't decided on the details

[STH: conclusion/resolution: no new bit of state (IOW, I agree with WH)]

Various Oddities/Backward Compat Issues, 1

…Missed first slide…

Various Oddities/Backward Compat Issues, 2

* Array.prototype.concat
  - Currently always creates Array instance, for subclasses usually want subclass instance

    - Change to use subclass constructor to create subclass instances but only when this object is tagged as array subclass
  - Currently auto-spreads Array instance arguments
    - Similar to above, auto-spread tagged array subclass args.
* Must compatibly support this idiom:
    - `[].concat.apply(Array.prototype, arguments);`
* Precomputing result length will support use with TypedArrays
* Similar result object handling for slice, splice, map, filter(?)

(Re: Bullet 1)
WH: Concerned about the proliferation of "types of an object"… class, prototype, type… species?

(Re: Bullet 2)
Discussion focuses around built-ins whose prototype property are first-born instances of that object, eg. `Array.prototype` is an `Array`. The desire was to remove this from ES6, however there a slew of issues that are not specifically edge cases rise to the surface.

Conclusion/Resolution, Bullet 2
Brendan to experiment with changing the prototype properties of Boolean, Number and String away from first born instances into ordinary object ([object Object]).

(Re: Bullet 4)
Discussion focuses on what type ("species") of object should be created by these methods when subclassing.

AWB: All of them, "species" is what you want, except in the case of map. For slice, splice and filter, you always want to operate on and return the "species" calling.

?: Want to customize it differently for different methods.

WH: (Doesn't want another "type" thing)

YK: (Doesn't think it's a problem)

AWB: You can always cast your object back into the type of your class with `this.constructor`.

Illustrating Bullet 4:
```js
class V extends Array {
  constructor(...args) {
    super(...args);
  }
}

var v, m;
v = new V(1,2,3, );
m = v.map(val => val * 2);
console.log( m instanceof V ); // false :(
```

Various Oddities/Backward Compat Issues, 3

* String.prototype.match, replace, search, split.
  - Currently spec'ed to directly use RegExp internal APIs which limit the ability to use them with RegExp subclasses that use alternative engines that don't expose those APIs
  - Refactor into public operations upon RegExp subclass

- String methods delegate to RegExp methods

MM: Confirm that any symbols defined only need to be unique, not private?

AWB: Yes.

4.10 Data collection/analysis: Function-in-block, duplicate parameter names, const, strict mode, var let/let[x]=1, and maybe more
(Presented by Brian Terlson)

Slides (pdf):
http://wiki.ecmascript.org/lib/exe/fetch.php?id=meetings%3Ameeting_jan_29_2013&cache=cache&media=meetings:real-world-js-code.pdf

1.85% of scripts make use of function hoisting intersection semantics. Almost all of these are due to functions defined inside arms of an if statement. Few examples of defining functions inside loops.
No scripts do let[x].
10% of "use strict" directives are incorrect because they're not in the prologues.

Mixed discussion regarding the actual expectation of these cases.

Can the tests be expanded beyond the top 10k?

BE: w/r to FIB, should we go with strict mode standardization?

YK: Does it always leak the identifier?

MM: In different ways.

Discussion regarding language forking.

LH: (Identifies a dark future that hinges on "use strict")

MM: That should be the future.

…
YK: Are there intersections of existing implementations that can be spec'ed?

Mixed discussion.

Table discussion until tomorrow.

 # January 30 2013 Meeting Notes

John Neumann (JN), Norbert Lindenberg (NL), Allen Wirfs-Brock (AWB), Rick Waldron (RW), Waldemar Horwat (WH), Tim Disney (TD), Eric Ferraiuolo (EF), Sam Tobin-Hochstadt (STH), Erik Arvidsson (EA), Brian Terlson (BT), Luke Hoban (LH), Matt Sweeney (MS), Doug Crockford (DC), Yehuda Katz (YK), Nebojša Ćirić (NC), Brendan Eich (BE), Andreas Rossberg (ARB)

Function In Block Options

(Presented by Luke Hoban)

Option #1: No let and const in non-strict mode
Option #2: Taking the breaking change, w/ evangelism
Option #3: Hybrid semantics for function-in-block, that still maintained semantic alignment with let and const
Option #4: let/const anywhere in scope changes FIB semantics.

LH: a combination of 1 & 3 might be the best path forward.

Option #3

```js
function foo() {
  bar();/// throw, extended TDZ
  for (var i = 0; i < 10; i++ ) {
    bar(); // throw, TDZ
    let x = 10;
    // semantics of function in block are as though...
    // 1. let bar = function() {} were inserted
    // at start of block
    // 2. var bar = function() {} were inserted
    // at start of function
    // (1) accomplishes ES5 let/const semantics,
    // (2) accomplishes compat
    // These are combined with an extension of the TDZ rules
    // to throw on accesses to let bound variables
    // whose activation block has not et been created.
    function bar() {
      console.log(x);
    }
    bar(); // succeed
  }
  bar(); // throw, eTDZ
}
```

WH/ARB: if there is a let or const anywhere in the program, don't do this in sloppy mode?

LH: There is an option, if you have any let or const, then function block is ES6 semantics... But I don't know how we'd define this.

YK: I want to go on record that I don't support this kind of unrelated opt-in.

RW: This is a language level weak inference...

WH: Far less a concern than burning people for the next twenty years.

BE: It's not an opt-in, it's allowing the syntax to speak for itself.

Discussion re: ES6 static semantics to determine runtime semantics.

LH: Option #4 is a possibility

MM: For both strict and non-strict?

LH: Option #4 in sloppy mode and clean up in strict mode.

...Key thing to note: the extended TDZ.

AWB: Is it compatible semantics with whats observed on the web...

Discussion about compatibility of options

STH: Pick one whether it works with what we want to do later

AWB: What are the compatibility requirements?

Discussion re: the sanity of each implementation's semantics.

BE: Not sure any are worth rescuing in sloppy mode. Leaning towards strict mode to align the let, const and fix FIB semantics.

LH: (Presents: http://web.archive.org/web/20120513130620/http://whereswalden.com/2011/01/24/new-es5-strict-mode-requirement-function-statements-not-at-top-level-of-a-program-or-function-are-prohibited/)

Mixed Discussion centered around the intersection of implementation semantics. Need to avoid creating yet a _new_ set of semantics that are outside of the intersection semantics.

Avoid specifying any of this in sloppy mode? BE, MM, ARB, WH agree.

LH: I still think there is something that we can do to make this work.

MM, BE: Avoid two identities for function declarations.

AWB: Reiterates proposal: Don't hoist the initialization of the function, initialize at the point of declaration.

STH: 1) define the semantics of let, const and FIB in sloppy mode and require special casing, 2) don't define let, const and FIB in sloppy mode. Define how the interact that's "mostly compatible with the intersection. Or we prevent them from interacting by ruling out sloppy mode.

BE: ES6 let in sloppy code makes FIB ES6 semantics

Option #2 will be rejected by implementors

Option #1 No let and const in non-strict mode

LH: Addresses compatibility concern, Forks the language.

Paths of teaching JavaScript will hinge on whether you include "use strict".

eg. Can't copy-paste code from blog post into script tag.

MM, WH: Would like a meta tag that makes all scripts and inline JS strict-mode.

LH: Recognizes the ease of copy/paste JavaScript for teaching purposed.

MM: The future will all be "use strict"

RW: In the last two days there has been two different blog posts focused on _removing_ strictness from JavaScript.

YK: Even if you think strict mode is ok now, but still has low uptake. We're making it an obligation for new features which is too large of change.

Mixed discussion about the semantics and restrictions of strict mode.

Further discussion about how simple examples and teaching JS in the future will require "use strict". Recognizing the real consequences of copy/paste restrictions.

LH: Concerned that Option #1 will absorb ALL new things.

Options

Option #0: No new syntax in non-strict mode
  0a. Truly no syntax (ARB) **Not Alive**
  0b. Allow module, but nothing else
  0c. Allow module and class, but nothing else
  0d. Allow everything with a body, but nothing else (the body is implicitly strict) **Not Alive**

Option #1: No block scoping in non-strict mode **No Support**
  1a. Preserves everything except let, const and FIB

Option #2: Taking the breaking change **No Support**
  - (w/ evangelism)

Option #3: Hybrid semantics for function-in-block
  3a. In both strict and non-strict **No Support**
  3b. In only non-strict. Strict keeps ES6 block scope functions
  3c. Absolute minimum intersection semantics supported in non-strict, else ES6 semantics

Option #4: let/const anywhere in scope changes FIB semantics

Option #5: No block scoping in ES6 **Strong Pushback (after discussion)**

(Note: "block scoping" means: let, const, class, FIB)

WH: Are the options listed about what we want to spec or what the implementations ought to *do* (leaving aside the question of whether or how the standard specs it)?
LH: Here we're explicitly discussing what implementations should *do*.
[general agreement that this is what we're discussing at this point.]

LH: We cannot do FIB with meaningful semantics in the face of let/const

AWB: If you didn't have FIB, are there still issues?

LH: No.
...To avoid any issues at all, the solution is Option #5.

RW: Does that also include "class"?

STH/LH: Option #3 is the counter point to Option #5.

LH: Option #5 is only legit if Option #3 is rejected.

BE: Option #5 means that class can only appear where functions can appear.

MM: (reiterates belief that one the experiment of extending ES6 features into sloppy mode has failed) ((Note to Rick: We repeatedly clarified that Dave's 1JS is not the extension of ES6 into sloppy mode. It is the earlier realization that we didn't need an additional mode switch, such as an ES6 pragma.))

AWB: Working towards balancing the future... future advantages of block scoping are so large that we should be willing to find a way to make them work.

BE, WH: The future is bigger then the past.

ARB: Wants #0 as cleanest and easiest path forward. (list updated)

BE: Option #0a is dead on arrival. (list updated)

AWB: Does #0d exclude rest and spread, destructuring? (list updated)

BE: Yes, Andreas and others discovered potential issues, probably not worth pushing.

...Yes, going with Option #0, changes the future

STH: One JS is about not having a switch to get new language features, eg. a language version etc.

RW: "use strict" is still a big red switch.

STH: Yes

BE: Committee opinion on #0d?
#0d "not alive" (list updated)

Circular discussion about macrocodes caused by Option #1

YK: What about Option #4?

ARB: No.

AWB: Option #4 is an extension of Option #3, because it allows you to safely shift into supporting let and const.

ARB: Strongly dislike because it has a refactoring hazard.

LH: If you're in strict, you already get correct semantics. Option #4 only applies to non-strict mode.

ARB: Want as few component interactions as possible.

WH: This is the closest we can get in the face of web compatibility.

AWB: Proposes Option #3c...? In only the specific cases where a reference does not have a local declaration

WH:

AWB: Willing to throw out function declarations created in eval.

WH: (Whiteboard)

```js
var f = ...
function a() {
  if ( false ) {
    function f() {}
  }
```

```
  f();
}
```

WH: What do we want this to do?

BE: In IE, the inner f hoists. In SpiderMonkey, no hoist. What should we do?

Option #3c: Absolute minimum intersection semantics supported in non-strict, else ES6 semantics.

BE: I think we can do #3c and Allen can make it work, concerned that Waldemar will poke holes and be unhappy.

WH: The above must call the outer f in strict mode. In non-strict mode we shouldn't specify what this does (it becomes a moot point if we continue to outlaw nested functions in non-strict mode). For compatibility we'd want to add an informative note stating that, if the implementation extends non-strict mode with nested functions, the similar program below calls the inner f:

```js
var f = ...
function a() {
  if ( true ) {
    function f() {}
  }

  f();
}
```

...There is no support for #3a (list updated)

LH: #3b is the bridge between compatibility and progress.

AWB: #3c covers the minimum use cases for compatibility

WH: #3c and 4 are similar

LH: Still supporting #3b

(Break for Lunch)

LH: Summary of current discussion. Why is that people are concerned about any of the #3's?

ARB: A lot of time for the committee to spend on identifying the sloppy mode issues. Why introduce complexity for something "transitional"

LH: So, not really an argument _against_ #3, but for #0

ARB: And not creating a more complicated language

AWB/EA: No.

AWB: [#3] Arguably no more complicated then [#0]

MM: (Uses "duplicate parameters in parameter lists with defaults" as example of adding complexity)

AWB/MM/WH: (Discussion around complexity implications)

AWB: Complexity that occurs in edge cases is not

LH: We can't argue complexity... Let's look at the trade off for long term goals of the language

BE: Can we get rid of Option #1?

Yes. Option #1: no support. (list updated)

LH: For those that support Option #0, If we couldn't get consensus on Option #0, what are the material arguments against Option #3?

BE: (Makes further case for the future friendliness of #0d)

LH: Reiterates the previous question...

BE: Option #3c could work really well

WH: If we go with #3c, there is no spec work, just implementors

AWB: Yes, there is spec work...

Discussion comparing the specifics of #3b & #3c

Support moving to #3c.

AWB: If sloppy mode, here is a compatibility issue.

MM/ARB: (Answering Luke's earlier question) If Option #0 is off the table, then there is no opposition to #3b or #3c.

Consensus
Allen will spec #3c static semantics with informative note for review, tentatively to fallback to #3b

BE: Make an error?

MM: Benefit to future implementors that want to be in conformance by making them errors?

BE: Refutes

MM: Retracts

AWB: Sloppy mode has let, const and class grammatically and semantics

Return to the bigger issue...

Reiterates Kevin Smith's proposal:

1. No opt-in required for new syntax, except
2. Except: No breaking changes to sloppy mode
3. No grammar contortions (e.g. let) to support sloppy mode
4. All new syntax forms with code bodies are implicit strict

BE: Appears to be stand between #0 and #3

MM: Still opposed to the let issue, by creating a potential hazard.

BE: At last meeting we agreed to break this... (whiteboard)

```js
var let;
let[x] = y;
```

AWB: We discussed this last meeting and agreed.

MM: I changed my mind.

WH: This is actually a grammar issue, which means that you must be able to parse it before you know whether or not you're in strict mode.

MM: Reiterates concern about creating potential hazards

LH: If we want to add new syntax, we're going to encounter these issues no matter what "mode" we're in... Hinging new syntax the fact that a small window reserved a few keywords won't scale (let, const, etc).

AWB: There is a certain circularity in arguments that are fine with new features only in strict, but also fine with allowing a single conflicting

MM: Reiterates

WH: So the argument is that we're making "sloppy mode" too attractive?

MM: Yes, avoid linguistic screw-ups.

LH: A lot of us don't view as "screw-ups"

WH: strict mode is just too complicated for user land code with many scripts on one page.

MM: If we're going to continue to support "sloppy mode", we need to make any use of "let" as a variable an error (retro-active reserve?). Cites Safari experiment of "let", Firefox experiments with "let".

LH/BT: Get data on use of "let"

MM: I'm back to where I was previously... I don't care enough to argue anymore.

(Note: Previous discussion can be found here: https://github.com/rwldrn/tc39-notes/blob/master/es6/2012-11/nov-29.md#the-syntax-of-let)

LH: Option #0 is too short sighted and based on "luck"... There won't be another opportunity to retro-reserve words, as there was with Strict Mode

MM: Two scenarios:
1. A breaking change that leaves the language in a better, more consistent state.
2. A breaking change that mars the languages and creates something that you constantly need to be aware of...

LH: (anecdotal C# experience)

MM: Counter with sentiments re: language complexity.

AWB/WH: Appear to be converging...

MM: I won't block the let kludge

AWB: But we need to address the larger issue of features in non-strict mode

WH: If there are concrete concerns, I want them on the table, but I don't want to spend more time on abstract arguments.

ARB: Concrete issues have arisen

WH: What are they?

AWB: We expect to have concerns.

ARB: issues with destructuring in sloppy mode

AWB: We have a spec for this.

MM: A spec that special cases for these issues and pays an unnecessary complexity cost.

AWB: Reiterates the concern that user code will not want to "use strict"

EA: I don't want forked languages

ARB/MM: Too many small breaks add up, I want to avoid introducing new breaks.

AWB: Fundamentally two approaches:
1. A discontinuity, resulting in multiple versions.
2. A single version, evolves incrementally.

LH: (Insight regarding the effects of "mental forking")

WH/DC: (Discussion about the inline-script problem)

DC: Code has long been moving away from inline code and our goals shouldn't be to preserve that.

LH: But there are still multiple scripts

DC: Strict mode is the path forward, we can maintain ES3 and ES5 "sloppy mode", but ES5 "strict mode" is the way forward.

AWB/YK: Even the term "sloppy mode" is a pejorative.

DC/YK: (Discussion about the current state of strict mode)

WH/YK: We have data that 10% of uses aren't even being used correctly, where "use strict" isn't correctly in the prologue position.

YK: Worried that people will want to use new features, but simply won't understand the requirement "use strict"

DC: This is not hard

...Continued discussion.

LH: ...Interjects to resteer conversation

?: The fundamental disagreement is about whether we want to make a simpler spec for the future by basing new features on strict mode only.

WH: No, that's not the bone of contention. WH would also prefer to have a simpler strict-only spec for new features, but supports 3c/4 from earlier because getting into strict mode is too inconvenient. Today it's too awkward to get into strict mode in practice for web pages, although this may change in the future?. Too many (10%) web pages unknowingly get this wrong by putting a "use strict" somewhere other than the preamble.

MM: What do you see would need to happen to make strict mode convenient enough?

WH: Two requirements:
1. Need a clear indication that a user's attempt to get into strict mode failed, such as having a ```use strict``` directive rather than overloading a string constant. It's obvious from the data that the bug that the strict mode directive doesn't go into strict mode if it's in the wrong place hits too many users.
2. Need a way to declare that all scripts on a page are strict. That's out of our control, but this is a prerequisite to making strict mode convenient for multi-script pages (and WH is aware of the potential issues of scripts coming from different sources).

YK: Hopefully I've made it clear that the current state of strict mode tooling is at a significant disadvantage.

MM: I understand that we need to work with DOM spec writers to get means of making "global" strict mode a reality.

Revisiting past conversations that led from ES6 being "versioned", to being "strict mode" only, to being "one js".

LH: We did adopt One JS. There have been two concrete issues:

MM: Terminology: "one js"?

LH: As in, new features do not require "use strict" or a "version"... Recommend continuing in the default direction: Continue on the path we're taking.

MM: That is not the meaning of "one js" (Rick: It's really important that we not confuse this issue with 1JS.))

AWB: If we can't agree on a new position.

WH: Wants to hear from others...

RW: (big spiel about strict mode and new features)

EF: Agrees and supports default strict mode in modules. Not enough of a historic sample to say whether what size "switch" and how many "switches" to have will be idea. We only have one real sample so far, ES3 -> ES5. The new ES6 strict mode will only be the second.

RW: Agrees with Eric and supports strict bodies by default.

DC: Including class?

RW: Yes

YK: Not Arrows

MM: Agree, not on that boundary.

LH/MM: Closer to comfort if classes and modules have strict bodies.

MM: Agreed not to hold up consensus

YK/RW: No Arrows

Conclusion/Resolution
Consensus on...
Stay the course on spec development approach
Class, Modules implicitly strict.
Arrows not strict
Sustaining our position on the handling of let ambiguities
(Note: this is a local consensus)

MM: Declared as not holding up consensus, though not in agreement. I still think this is a terrible idea, but it looks like the best we can declare consensus on.

BE: (Out of band) I think that's a good consensus, arrows can't have intrinsic names (contrast with NFEs) so arguments.callee may be wanted, so arrows should not be strict-only.

Static/Class Side Methods
(Presented by Yehuda Katz)

```
ClassElement:
  MethodDefinition
  "static" MethodDefinition
```

Full strawman here: http://wiki.ecmascript.org/doku.php?id=strawman:class_method_syntax

Supported by TypeScript, Continuum, etc.

Conclusion/Resolution
Consensus in support of this proposal. (File spec update ticket)

Revising The Array Subclassing "Kind" Issue
(Presented by Allen Wirfs-Brock)

Reiterates the issue at hand, illustrated here:

```js
class V extends Array {
  constructor(...args) {
  super(...args);
  }
}

var v, m;
v = new V(1,2,3, );
m = v.map(val => val * 2);
console.log( m instanceof V ); // false :(
```

```

If we extend...

`Array.from( iterable ) => Array.from( iterable, mapFn )`

```js
// Add a "map" function to the class-side from method:
NodeList.from( nodelist, thing => thing );

// Turn an array into NodeList
NodeList.from( array, thing => thing );

// Turn an array of nodeNames into NodeList of nodes
NodeList.from( ["div"], node => document.createElement(node) );
```

MM: thisArg?

EF: It's possible to lose the context if you alias the ClassMethod

MM: For consistency purposes, add thisArg

Conclusion/Resolution
Consensus on this proposal, with the addition of thisArg per Mark's request.

4.7 Fail-fast destructuring with ?-syntax for irrefutable opt-in
(Presented by Brendan Eich, Andreas Rossberg)

1. No ToObject(RHS).
2. Exception on missing property selected without a ?-suffix.
3. ?-suffix allowed on any pattern, imputing undefined deeply instead of refuting.
4. The ? is a separate lexeme from the : in long-hand patterns.

One entry point:
https://mail.mozilla.org/pipermail/es-discuss/2013-January/027800.html

Allen argues cogently against (1) but endorses (2-4).

BE: (Whiteboard)
```js
var { toLocaleString, split } = "";
```

The only ones that are affected: string, number, boolean.

ARB: implicit conversion is future-hostile to pattern matching.

STH: (Whiteboard) Example of pattern matching.

```
switch ("x") {
  match {}: return 1;
  match _: return 2;
}
```
```

You do not want implicit conversions in pattern matching, which would make this take the first branch. Similarly, if we allow matching literal strings.

ARB: For consistency, would imply conversions for other pattern types, too (e.g. literals). No issue technically, but insane from practical perspective

BE: Reiterates... In the case where

There is no history of primitives being used on the RHS, so there is no need to support any kind of implicit ToObject.

"You say po-tah-to and that doesn't change tomato".

...

MM: Question about cover grammar.

AWB: The grammar is spec'ed and in the latest draft.

...

YK: Wants irrefutable destructuring, ie.

```js
function Foo(options) {
  var { foo } = options;
  // if foo is not a property of options,
  // just give me "foo" that's undefined
}
```

BE: Use the ?

```js
function Foo(options) {
  var { foo }? = options;
}
```

WH: Is ? deep or shallow? i.e. what should the following do
var { p, r }? = q;
? answered:
- if q is null: p and r are undefined
- if q is {r: 7}: p is undefined, r is 7.
WH: That means that ? is deep.

WH: Given the discussion above [not recorded in these notes?] about doing primitive pattern matching assertions in the future, just curious what the semantics of ? ought to be so that it would interact well with such assertions. Example:
What should ? do in these cases?
let {p, q}? = {p:17}
Here p gets 17 and q gets undefined because ? distributes down.
But then what happens when we get value assertions? Without getting into syntax details, assume that x: true is a value assertion that fails unless property x exists and has the value true.
let {p: {x: true}}? = null
let {p: {x: true}}? = {q: null}
let {p: {x: true}}? = {p: null}

let {p: {x: true}}? = {p: {}}

Mixed Discussion and debate, re: pattern matching examples.

BE: Return to... Waldemar, are you satisfied with ? for irrefutable destructuring?

WH: The question, is there a purpose of having a shallow "?".

BE: The common case is implied deep "?"

AWB: More comfortable if there was a strawman of what the full language would be like in the future.

BE: Dave has a proposal written, but we need to capture the latest developments.

ARB will work out new pattern matching strawman based on proposed ideas.

BE: (to YK) have we assuaged your concerns about irrefutable destructuring?

YK: yes, as long as "?"

Conclusion/Resolution
1. No implicit ToObject(RHS). **consensus**
2. Exception on missing property selected without a ?-suffix. **consensus**
3. ?-suffix allowed on any pattern, imputing undefined deeply instead of refuting. **consensus**
4. The ? is a separate lexeme from the : in long-hand patterns. **consensus**

"?" in combination with default values, syntax error.
```js
{ a? = 42 } // throws.
```

If you write a "?" on a larger structure, you can have default values anywhere inside.

```js
{ a: b = 42 }? = undefined
```

WH: Where should the ? go syntactically? In particular, should we have
{p: q?}, {p:? q}, or {p?: q}

STH:
```js
let { p }? = null;  // p bound to undefined
let { p? } = null;  // throws
let { p: q }? = null; // q bound to undefined
let { p?: q } = null; // throws
```

WH: This is quite close to conflicting with the ternary operator but can't think of any obvious ambiguities as long as the ? in patterns is always followed by =, }, comma, or :.

Dependent on patterns being fail-fast:
ARB: Cute idea: Make ```undefined``` into a keyword that is also a pattern that matches only the undefined value. This way existing definitions such as
var undefined;
and

var undefined = <something that evaluates to undefined>;
and
(function (undefined){})()
will work, but
var undefined = 42;
will fail.

General Agreement

4.11 Name Property of Functions
(Presented by Brendan Eich, with additions by Brandon Benvie)
http://wiki.ecmascript.org/doku.php?id=strawman:name_property_of_functions

BE: Early approaches...

```js
function f() {}
f.name == "f";
```

BE: (from Brandon Benvie's proposal)

Justification: the usefulness of the name of function is not just for debugging. It is useful in the same ways that property names as strings are such as dispatching by name or assigning by name.

1. Every function has an own "name" property, and this property is always a string (unless the user specifically decides to violate this norm).
2. This name property is initialized with the value that makes sense from static semantics.
3. Allow predefined names to be altered in cases where it makes sense.

Semantics:

The baseline descriptor for every function is the 'name' property defined as
{ value: "",
  writable: true,
  enumerable: false,
  configurable: false }

For FunctionDeclarations, named FunctionExpressions, MethodDefinitions, or accessor Properties then the function's "name" property is set to the given identifier.

In the case of the constructor method of classes, the class
name is used instead.

In the case of accessors, "get" or "set" is included.

The "name" property is set to non-writable.

Function.prototype's name is also non-writable.

Anonymous FunctionExpressions and ArrowFunctionExpressions assigned in a VariableDeclaration or ObjectExpression are given the name of the variable or property they are assigned to and the name remains writable.

Anonymous ClassExpressions follow the same semantics, with the name being used for the constructor method.

Whenever a function's name is defined by a Symbol instead of a regular identifier then the name is the result of ToString(symbol).

The name property should (probably) not have any reflection on the output of Function.prototype.toString.

MM: Having the result of toString be immutable is important, agree with proposal semantics.
...Security leakage from the name property, is the type of information that the toString method leaks. No integrity issue, just a confidentially issue.

BE: We _could_ make Function.prototype.toString be a getter that can be removed.

MM: This is one of the aspects that I gave up attempts to secure, and it's never been an issue.

BE: Make it writable?

WH/MM/AWB/YK: Writable.

WH: There is a slight advantage to making it non-writable. Otherwise when writing it to change the function name, you'd expect it to change the result of "toString".

Discussion re: history of Function.prototype.toString result.

MM: Explain the divergence, toString is a reflection of the source code itself and the name is how you'd like to identify the function within tools.

Conclusion/Resolution
Consensus on the proposal, with { writable: true }

4.12 Typed Arrays Update
(Presented by Allen Wirfs-Brock)

Slides (pdf):
http://wiki.ecmascript.org/lib/exe/fetch.php?id=meetings%3Ameeting_jan_29_2013&cache=cache&media=meetings:typedarray_status.pdf

AWB: We're absorbing control of the TypedArray specification, previously authored by Khronos.

W3C has dependencies on the Khronos spec.

This specification will not include any parts of the Khronos spec, but will be compatible.

JN: We need to make sure we're not violating anyone's IPR, Istvan can inquire.

EA: We can have Ken (Kenneth Russell, Google) review the drafts.

LH: Short term, the editors need to understand what's going on and there is no push back. The standards organizations need to know that we're now working on this specification. **Agreement**

AWB: (Proceeds to technical presentation)

Integrate into ES Spec
- ES spec conventions and semantics not WebIDL
  - Khronos spec not necessarily tracking WebIDL

  - eg. instanceof
- Lot's of implementation differences among browsers at MOP level to straighten out
- TypedArrays are subclassable

LH: TypedArray implementations are incompatible in the same way that all DOM implementations are incompatible, because WebIDL is not implemented compatibly.

Max Length
- Currently Khronos spec's all lengths as Uint32
- Not future friendly, especially for byte sussed element arrays
    - eg. a Uint8Array might map to a large real memory-mapped buffer bigger then 4GB.

LH/AWB: Seems that WebIDL is the cause of this constraint.

AWB: I'm going to spec the max length at Int, not Uint32

LH: Should record the deltas

Khronos/W3C TypedArray Objects
(Diagram)

AWB: 9 prototype objects, 54 distinct method/get accessor functions per Realm.

MM: How did we get 9?

LH: UintClampedArray, which used to be CanvasPixelArray

Prototype Hierarchy Factoring
(Diagram)

AWB: 10 Prototype objects, 6 distinct method/get accessor functions per Realm.

#10, TypedArray.prototype
+ [BYTES_PER_ELEMENT: int abstract]
+ set() void
+ subarray() void
+ byteLength() int get
+ byteOffset() int get
+ buffer() Object get
+ length() long get

Int8Array (BYTES_PER_ELEMENT int=1),
Int32Array (BYTES_PER_ELEMENT int=4),
Uint8Array (BYTES_PER_ELEMENT int=8),
Uint32Array (BYTES_PER_ELEMENT int=4),
Float32Array (BYTES_PER_ELEMENT int=4),
Float64Array (BYTES_PER_ELEMENT int=8),
Int16Array (BYTES_PER_ELEMENT int=1),
Uint16Array (BYTES_PER_ELEMENT int=1),
Uint8ClampedArray

MM: A Uint1Array or BooleanArray, where each element is one bit.

TypedArrays act like fixed length, numeric element JS Arrays

- So why not even better Array integration?
- Class methods?
  - TypedArray.of
  - TypedArray.from
- TypedArrays should be iterables?
  - @@iterator
  - keys
  - values
  - entries

Even Better Array Integration
- Other Array.prototype methods that will work just fine on TypedArrays
  - toString, toLocaleString, concat, join, reverse, slice, sort, indexOf, lastIndexof, every, some, forEach, map, filter, reduce, reduceRight
- Only 5 Array.prototype methods won't work with TypedArrays
  - push, pop, shift, unshift, splice

Add Array methods to TypedArray
(Diagram)

WH: Are TypedArrays spread by Array.prototype.concat and ...?

LH: TypedArrays are a targeted tool for byte level programming.
...Doesn't think that the generic, higher-order Array APIs should be exposed on TypedArray

WH: [Repeats question]

?: TypedArrays are not exotic arrays so don't get spread by Array.prototype.concat and ... . On the other hand, TypedArray.prototype.concat will spread TypedArrays only.

WH: OK.

[Debate about whether TypedArrays need the suite of Array methods]

YK: Using TypedArrays would be a pain if they didn't already offer these array operations.

WH: It would be too confusing to have gratuitously different APIs for two array-like things in the language.

RW: Agree.

AWB: My understanding was that we were absorbing these in order to provide the capabilities that the language provides

STH/WH/YK/RW/EA: Strong Agreement.

LH: Agree that there are some use-cases for these, but may not be our place to define these methods

YK: (channelling Alex Russell) The reason we're taking this on, is because it's our responsibility to correct "yet another array like thing that isn't an array"

RW: Agrees.

LH: Implementation and performance issue concerns?

WH: This is not a lot of work for implementations, and implementations (as opposed to users) are clearly in the better place to implement these efficiently.

[STH: After the end of the official meeting, there was some discussion as to why Typed Arrays wouldn't just become genuine Arrays. This would mean
that they'd have all the Array methods, some of which would throw on use (those that change length). This needs discussion at a future meeting.]

Conclusion/Resolution
Move forward with the plan presented.

 # January 31 2013 Meeting Notes

John Neumann (JN), Allen Wirfs-Brock (AWB), Rick Waldron (RW), Waldemar Horwat (WH), Eric Ferraiuolo (EF), Erik Arvidsson (EA), Luke Hoban (LH), Matt Sweeney (MS), Doug Crockford (DC), Yehuda Katz (YK), Andreas Rossberg (ARB), Sam Tobin-Hochstadt (STH), Brendan Eich (BE),

4.14 Module Update
(Presented by Sam Tobin-Hochstadt)
TODO: Request Slides

STH: (Slide 1)

Recap from November

- Modules no longer named by Identifiers.

```js
module "m" {
  import "x/y" as y;
  export y;
}
```

DC: Are they strings?

STH: They are string literals, but may have an internal structure. We should allow characters that don't have to be escaped in URLs.

LH: This is something we will definitely have to specify.

STH: Confirms that they are string literals.

DC: Why strings and not Identifiers?

STH: The string indicates a different scoping behavior

MM: Nesting example?

STH: (Slide 2)

Nested Modules

```js
module "outer" {
  let y = 1; // not exported
  module "inner" {
    let z = y + 1;
    export z;
  }
}
import {z} from "outer/inner";
```

MM: Why isn't "inner" exported?

YK: All modules are visible/available.

MM: Why are all modules visible?

Concern about export visibility.

STH: You're both actually in agreement

LH: There was a change from a more complex system, now when you write something inline, it's loaded into the internal loader registry. This simplifies the proposal, removing the "hybrid" behavior of an Identifier.

STH: Believe this is a better design.

ARB: Disagree.

MM: Q. Where the "inner" string appears, there is no way to change the nesting path.

ARB: ...

MM:

AWB: When you do an import, is that passed through the Loader?

STH: Yes.

AWB: So you could write a Loader that blocked "inner"?

STH: You could enforce any arbitrary rules in the Loader.

ARB: What happens if inside "outer", I have "import from inner"?

STH: Won't work. There is a ...

EA: What happens if you do "import {a} from outer"?

STH: Don't understand.

STH: Important both have a mechanism for top level and relative,

ARB: Appears to be reinventing half of lexical scoping in an ad-hoc way, super-imposed on strings.
When somebody thinks to have a better system than lexical scoping, they usually turn out to be wrong.

STH: Trying to avoid this being like lexical scoping. The goal is to be explicitly different, closer to Node... refers to require()

YK: There is a large data set supporting

LH:

YK: Mentions concatenation argument, determined as not strong enough.

STH: ...gave an example of a module that may want nested modules...?

YK: Predict it to be multiple files, not sub modules within a module

STH/YK: If you have a top level...

AWB: clarify? you can't take two top level modules, concatenate them and put them in a script tag?

STH: Yes, but you can't import from them.

LH: Clarifies.

AWB: There isn't a transformation that would take all of the modules and concat into a single file and it would mean the same thing?

STH: You could certainly write a transformation, but you would have to put that result into something that

ARB: What if I import inner, but outer hasn't been imported yet?

STH: Outer is implicitly loaded.

...

YK: I previously understood the module system (STH: I thought it was EA who said this)

LH: I thought... There is a loader that has a table of things. Two ways to get them in there:
1. Filesystem, where keys are the path
2. Load a script

Are you saying that the file name is now unnecessary?

STH: Yes

LH: I don't see how you could make this work

YK: I think it's not good. You can't dictate that file A must be _here_ for file B to work.

AWB: You don't want to mix physical identifiers with logical identifiers

ARB: Exactly

STH: One of the other ideas is that the Loader has setters, so that we can replace them... allowing you to change the default Loader. We are going to provide the basic programmatic layer, not design the declarative loader mapping mechanism.

YK: The requirement, based on the example:
"jquery-ui" needs "jquery"
...will be problematic when a developer cannot control where the files go.

MM: Will inner be executed?

STH: inner is not executed unless imported.

MM: is the identifier after the module keyword required?

STH: the semantics would likely involve immediate import of that module.

LH: I don't understand why we want nested modules like this. There is no existing system that supports the need.

YK: Agree.

STH: (whiteboard)

```js
import {$} from "jquery"
```

What does "jquery.js" look like?

YK: Top level $ with no "module {}"

STH: What if that file wants to use some other module? I think people will get that very wrong.

AWB: (Concern about the physical file and the import) Not operating logically at the same level as modules

EF: Agrees.

...Discussion regarding "loading" the physical and "importing" the logical.

LH: What Sam is saying... "why can't it just be the import syntax"

MM: Given a fully qualified path and module name, that uniquely identifies a module?

STH: Yes.

MM: What does TypeScript do?

LH: TypeScript has two notions of modules. "declarative object", similar to the earlier module proposal. Also has "file modules" that is a file with "module" that becomes an AMD define.

ARB/LH: Discussion of the semantics of the TypeScript type system.

STH: It assumes that all of the bindings are there for all time.

LH: Yes.

STH: I will avoid "possible futures". To re-answer, module sources are contiguous.

...

STH: Originally, Dave and I punted on nested modules, until we ran into a case that needs nested modules.

ARB: The result is a reinvention of lexical scoping of sorts.

...

AWB: Imagine you had a module named "." and you had a module in the same file named "./util" and so on... Then you take that and put it in a file named "jquery" and try to import from "."...

STH: What that suggests is that naming something "." will give strange behavior.

AWB: It represents an example of the problem you're against.

EA: If you import a file, called "jquery" and that has a nested module named "util", should work the same if "util" is just in the same directory.

Mixed discussion, re: what _could_ be solutions of

STH: Recapping... Yehuda and Erik brought up a concern about defining multiple top level modules and how to indicate that they are belong at the top level.

YK: another example, Ember relies on Handlebars...

STH: (whiteboard)

Solution?
```js
module "/jquery" {
}
.
.
```

LH: One of the things the module system needs, to move forward, there should be 5-10 real world scenarios. Each scenario needs to be met and _recorded_, each time a change is made, it needs to be applied to each scenario to ensure that we're covering the needs.

AWB/YK/RW: Strongly agree.

STH: Scenarios

1. Script tag, local file
2. Script tag, cdn
2. Inline

AWB: An over simplification

EF: Will always need to provide some kind of configuration, because in reality, files and libs have relevant version numbers.

YK: ...gives example for RSVP, Ember, Metamorph

EF: It should be clear how you link this name to a resource.

STH: Should be able to set up a loader initially and then "import" just works.

YK/STH: But we're not going to define the mechanism, because it's out of scope.

STH: Do you all understand why I don't think this committee should design a complete API for Loader?

EF: But it results in magic and Allen is talking about removing the magic.

STH: But that's configurable

EF: Configuring is not pleasant.

STH: `System.fetch(...url)`?

EF: Imagine if jQuery home page said: `System.fetch(jquery); import {$} from jquery`.

YK: We want a programmatic hook, but I want to say "import jquery" and I want to tell you where it is.

STH: This is easy to add, but I'm not sold. I will discuss this with Dave

YK: Is because you don't see the burden?

...The current proposal won't work when you want to use a CDN url and also have jQuery at the top level.

The primary issue:

```js
import { jQuery } from "jquery";
```

Needs to have some mechanism by which we can configure WHERE "jquery" is, via URL.

STH: A possible solution?
```js
Loader.configure([
  [ "jquery", "file.js" ]
]);
```

EF: The original proposal, where the file is inferred by the name doesn't get us close to meeting the need.

BE: This is not design by committee, we _really_ need user testing.

AWB: Are there people that are going to provide the specific scenarios that need to be met?

YK/EF/RW: (agree to work with Same and Dave)

STH: (slide 3)

Dropped Features
- export =
- export get/set

LH: Reminder to include the node use case in the scenario write ups.

STH: (slide 4)

Loader Semantics

- import foo/bar => fetches "foo/bar.js"
- import ./foo => relative to the file
- WIP, specifying the details of loading
    - Ordering
    - Error

- Hooks
- New "rewrite" hook

EA: Q. re #1, relative to what?

STH/YK: document, document base

ARB: Are you sure about "relative to file" on the second?

YK: This would break concatenation

STH: Important for self-containment

YK: The Loader has to know enough about the system to find the correct thing

Mixed discussion.

LH: We need to drive this through with concrete examples and use those steer the development here. I don't think we're going to get anywhere here today.

STH: (returning to the slide) Specified the "rewrite" hook to see that you want jQuery, it will fetch the source, eval it and then rewrite into jQuery.

DC: ...Recalling the original discussion around modules and the agreement he made with Dave to withhold objection on the grounds that we'd agree with syntax later. That agreement hasn't been met. The discussion we had this morning indicates that it has failed to meet the requirements and should not hold up the rest of the specification. Suggest to remove from ES6.

LH: As much as it would be great to deliver this, it's simply not ready and not nearly developed or concrete enough to continue forward with. We'd need a year to user test to get the best results from the feature. Need to understand and meet the needs.

BE: Library code can't dictate and may have to change, but that shouldn't block the feature.

STH: It's unfair to drop the feature when other features have behaved the same way and are still incomplete.

MM: To be fair, changes coming to proxy

STH: There is a great deal of completed work for this and with valuable feedback, we can still deliver. It's not fair to say "We're providing you with feedback, therefore you're not done".

BE: Yes, we need algorithms in the spec and user testing in time.

EA: Modules are the most important feature that we have to ship, but I agree with Doug that it might not be in time.

BE: One option is to move it to ES7

EA: From implementation, modules is the highest priority, but there is nothing to implement.

AWB: And much of the spec will rely on modules.

MM: Comparisons of the lexical scoped modules and the new string name modules

BE: ...Recalls the argument from 3 years ago. Notes that IIFE is NOT what `module {}` is and cannot be compared.

DC: I've spent years promoting this, but I can't see how it will make it.

AWB: We should aim for a "maximally minimal modules" proposal, that we can then move forward with to be successful.

WH: I like modules but also share Doug's skepticism about them being ready for ES6

ARB: I've been working on implementing modules for a year now and the changes you made in November invalidated most of that work, and they were not changes that I could agree with.

STH: To Clarify, we should not do a design along these lines AND you're concerned about the schedule.

ARB/AWB: Don't want to go back to ground zero.

LH: Start with a more robust design exercise, instead of trying to patch.

YK: To address the notion that the need has eroded, the systems that have been developed have actually put us in a worse position.

EA: Can we get champions of an alternate proposal? Andreas?

AWB: If we're going to defer or decouple modules from ES6, we need to know sooner, by the next meeting.

STH: I believe strongly that the state of the module system is being mis-perceived.

EA: I'm really confused by the current system and I feel like i used to understand the old module proposal well.

ARB: Same here.

LH: I don't think the current system is grounded or addresses the real problems that it needs to address.
...What we need to accomplish is much deeper.
...Prior art and experience dictates how the experience will be received and this doesn't seem to come up.

YK: I see modules as largely "desugaring" to two things that can be done with defining and loading in AMD, YUI etc.

LH: I'm not seeing all of the needs being met. My intuition is to say "this is a sugar for AMD", I think that could be a solid guiding principle.

YK: I agree that it's easier to see a path forward if it's largely the same thing as something that is already in use.

STH: re: possibly maximally minimal? There are too many details and requirements that are closer to surface, syntactically. It's harder to jettison the complicated parts to reduce the work, but also gives me confidence that the fundamental design is not changing as much as those in the room may belief.
...It aligned the programming model with the semantics of the Loader
...Believe it would be a big mistake not to do modules in ES6.
...Problematic that we're having this discussion without Dave present.

RW: We should break and table this until Dave can be present.

LH/DC/STH: Agree.

AWB: Concerned that we're on a path that won't include modules. If we're going to ship within a year of our target, modules won't make it.

STH: If we characterize it that way, then not many features are ready.

AWB: Agree.

RW: We should table this until the next meeting, where we are scheduled to actually discuss which features move forward and which don't. Additionally, Dave will be present.

STH: Of course over the next two months, Dave and I will be working with Allen to move modules into the spec.

EA: That could be the best thing.

RW: Likely to restore confidence if this can be achieved.

LH: Not convinced we have a design that is ready for the spec.

...Seeing this in spec may reify the unknown parts.

YK: If we don't have modules in ES6, certain parts will have to be respec'ed

AWB: I also believe a maximally minimal specification that can be used internally could happen.

LH: Of course, there is a status quo: Creating global names that define built ins

...Discussion re: Map, Set and WeakMap already in browsers.

JN: Steering to break. The point of record...

Conclusion/Resolution
- Cannot continue this discussion without Dave Herman
- Continue work, the understanding the next meeting is decision time.
- Work with committee members and community to develop concrete scenarios that must be met and addressed.

New Scope of the Royalty-Free Task Group (TC39TG1)

...Discussion...

WH: Motion

MM: Seconded

DC: Seconded

Conclusion/Resolution
We approve the new scope for RFTG Unanimously.

There was no specific agenda item here

ATTENTION: Mark, Waldemar, Allen... Please review these notes. I tried to keep up, but I'm not confident in the record.

MM: Want to maintain the option to defer Notification Proxies and Private Symbols—Defer to March?

...Mixed discussion about feature deferment.

WH: I agreed to classes based on private names inclusion, if there are no private names, then I can't agree to classes.

AWB: The objections have all been based on their interaction with Proxies

MM: I disagree, but don't want to deep dive.

WH: This is a bug in Proxy

AWB: Agree.

MM: I'd like to postpone a larger discussion until March.

AWB: (whiteboard)

```js
var p = Proxy(target, handler);
```

Any operation you do on "p" is forwarded onto "target"...

```js
var target = {
  foo() {
    if (someLookup(this)) {
    }
  }
};

target.foo();
p.foo();

// "this" in foo() is bound to the object
// SHOULD always be "target"
// EF: Could be multiple bindings for the same "target",
// so "this" should be "target"
```

AWB: Outlines issues with the MOP...

```js
target.[[Get]]("foo");
P?
self.[[Get]]()
...
```

See image.

target has a method named foo, uses the "this" value,

(WH's summary) The point that Allen was demonstrating on the whiteboard was that, given:
var target = {foo() {if (someLookup(this)) {...}}};
var p = Proxy(target, {});

then:
target.foo and p.foo are the same function object.
target.foo() and p.foo() get different this values (target and p respectively).

Use cases:

1. Virtual Object
- The target is not really the target, just a book keeping mechanism.
- Wouldn't need the target, despite need for invariant checks
- Nearly all functionality in the handler

2. Membranes (also, "nosuch...")

3. Caretaker
- Allows all through until revoked
- Nearly all functionality in target

AWB: A virtual object case... the target object is a captive object of each proxy, part of the implementation. Providing most of the functionality. The target never escapes.
...The case that doesn't cover that: the a proxy is being over laid on a pre-existing object.

YK/AWB: The case of defining an array-behaving-length object via a proxy on DefineOwnProperty works well.
WH: Is the pseudo-array example a virtual object or caretaker pattern?
AWB: Virtual object
WH: Why did you classify it that way? It's implemented almost entirely in the target, with but one method in the handler.

MM: target.foo() and p.foo() by themselves mean the same? Yes.
Are these still being spec'ed as a Reference Type? (base and a value).

AWB: When you evaluate p.foo(),
The [[Get]] operation returns ?
The [[Invoke]] operation returns ?

(TODO: Ask Allen to revisit)

MM:
[[Get]] value
[[Get]] base

AWB: In the spec, it's handled as a single operation...
A property access and then an invocation... we can separate

YK: This matches my misunderstanding.

AWB: We can trace this to Notification Proxies

MM: Recounts issues

Bookkeeping with invariant enforcement lead to exposure of the target.

YK: I want create a virtual object and you want to create a caretaker, might not be able to do both.

MM: Have to do both.

Issues with the current list of traps that can be resolved with the addition of a new trap that addresses... ?

AWB: Recommends a "call method" trap to resolve

WH: Still not sure what the motivation is. The behavior of target.foo() and p.foo() getting different this values (target and p respectively) seems correct. Turning function dispatch into a fundamental operation rather than a composite of a property lookup and call won't accomplish much. It would affect p.foo(), but wouldn't affect p.foo.call(p, ...), and I very much don't want to introduce another gratuitous asymmetry between the two.

MM: In the same way that [[Get]] was designed to make a decision, "Call Method"... (Mark, need clarification)

WH: Not specific to Call Method, but any case where you have self-referential object (i.e. any place where you do p.foo(..., p, ...)).

AWB: Anything that involves implicit "this"

AWB: If want to control what's passed through, you'd use a p.foo.call, otherwise you'd use a wrapped function.

MM: The default probably should be the proxy and not the target

```js
var q = Object.create( p );
q.foo();
```

MM: When you're inheriting from the Proxy, q is a normal object, does it have the property foo? it does not, recursive get on p and will get the proxy, q is the receiver.

WH: Call Method becomes useless...

MM: [q.foo] does the right [[Get]], `q.foo();` the value of "this" is q

...On the target, a getter for "foo", the handler trap is empty, uses the default, the getter function for "foo". What?

AWB: Unsure,

MM: The handler is given the ability to make a decision, same for Call Method. In the absence of a trap, what does the default do? The default case, "this" should be "q".

...The default, the getter should be invoked for "q.foo" case

AWB: ...Recalls discussion with Tom, re: [[Put]] creating properties on wrong objects.

STH: Resteer the discussion...

Conclusion/Resolution

Continue discussion offline.

MM: What we did with hasInstance, we should audit all of the traps and determine if we can replace with @@methods.

AWB: Applied similar approach in the spec.

Class extends: Throw on non-constructor?

In July, we discussed what would happen when:

```js
class Foo extends Object.prototype {

}
```

The decision of record is: Throw on non-constructor.

AWB: You may have class who's instances inherit on the instance side, not the class side.

LH: classes just simplify the Constructor pattern, and we should keep it that way.

RW: Can't you get the same semantics by omitting the super() call?

LH: No, you'll still get class-side.

...Discussion comes back around to throwing on non-constructor. It will also allow for future extension points by relaxing the RHS of extends.

LH/EA/RW/WH: In agreement.

Conclusion/Resolution
Class extends throws on non-constructor (extends null is still valid)

Map/Set comparator
(Presented by Allen Wirfs-Brock)

Issue: Allow Map or Set to have a custom comparator for set, get, has, delete

- Current constructor signature
  - Map(iterator=undefined)
  - Set(iterator=undefined)

- Proposed Signatures:
  - Map(iterator=undefined, comparator="default")
  - Set(iterator=undefined, comparator="default")

The Comparator Selector
- comparator must be one of the following values
    - "default" // The default comparator is used
    - "==" // == is used
    - "===" // === is used
    - "is" // Object.is is used

- The default comparator is the same as "Object.is" except that +0/-0 are considered equivalent and all NaN values are equivalent.
- Other values produce a range error. In the future we might allow a user provided object or function.
- Note that all the above comparators do object identity comparisons so map/set hash tables would identity based hashing (for objects) and as long as only the built-in map/set/weakmap hash tables are available there is no need to expose the interlay maintained per object hash value
- A future user defined comparator could be used for situations where there is a need to do nonidentity based comparison and ES code-level defined hash functions. Typically such comparisons/hasing would be based upon the values of the object properties. Such a comparator would probably have to expose both a comparison method and a hash method.

LH: Require a string value and throw on invalid

MM: happy with this proposal, any dissent?

WH: The comparators must be equivalence relations. Neither == nor === is an equivalence relation, with == breaking particularly badly. It's neither transitive, symmetric, nor reflexive.
AWB: You'd simply have unfindable elements.
WH: That might work for a multiset but not for something that claims to be a set. You could iterate through the keys and not be able to look up the corresponding elements. If you stick a NaN into such a map, you then couldn't delete it.
AWB: Reduce the set of comparators to "is" and "default".
WH: Rename "default" to "===" but keep its semantics (NaN's are equal, ±0 are not).
AWB, most: Yes.
YK: No. Maps should throw on NaN's.
WH: Why? Why not throw on undefineds?

Conclusion/Resolution
Consensus w/ comparators: "===" (currently default) and "is"

Signatures:
- Map(iterator=undefined, comparator="===")
- Set(iterator=undefined, comparator="===")

(Conversation continued, this resolution was discarded)

ARB: if we have a forth form of equality as "default", we should be honest, name it, and make it available separately

YK: Can we use "===" and disallow NaN as a key?

WH: No.

AWB: Why not?

Comparison of the "===" vs Object.is case, w/r to NaN, +0/-0

MM: Move `Object.is` to `Math.is`?

WH: No, it accepts non-numbers

WH: The longer we debate this, the more I want to just go with current implementations [that don't have a comparator parameter and always do the "is" semantics].

RW: +1

AWB: We should use this unnamed function as default, "default" is a fine name for it. If there is a demand, we can add it in the future.

EA: We can skip the name...

ARB: You might have circumstances where you do your comparison outside of the map.

MM: General comparator argument, are we happy with this considering a hypothetical future that allows arbitrary equivalence class?

WH: As long as it's an equivalence class, if you allow anything else, you run into trouble.

Conclusion/Resolution
- `Map( iterator = undefined, comparator = undefined )`
- `Set( iterator = undefined, comparator = undefined )`

- If the second argument is "is", use "Object.is"
- Otherwise, use "==="

```js
Map.defaultComparator( a, b )
Set.defaultComparator( a, b )
```

Comprehensions/Generator Syntax
(Presented by Sam Tobin-Hochstadt)

Dave Herman's proposal: https://gist.github.com/b250d1fad15dbb5f77a5

Existing:
```js
  let array = [x for x of itr if x > 0];
```

switch to
```js
  let array = [for (x of itr) if (x>0) x]
```

BE: ...Recalling that the draft proposal is right-to-left, this proposal is left-to-right.

...Mixed discussion with reminders of current specification.

LH: Would like to also have a keyword identifier to start the expression, maybe yield

BE: no, that must mean the outer function, not comprehension in it, is a generator, and would yield a directly enclosing function* or throw a SyntaxError otherwise.

BE: also, comprehensions must be lightweight syntactically or there' s no point.

WH: Expresses surprise at existence of parenthesized comprehension syntax: `(for (x of itr) if (x>0) x)`. This one returns an iterator? Somewhat unhappy that the base case doesn't work:
[x] returns a one-element array, but
(x) returns x, not a one-element iterator.

STH: This is only a syntactic change. We're not altering previously agreed semantics in any way.

Conclusion/Resolution
All agree on RTL -> LTR change.

[WH: Noted a moment after meeting adjourned that this is not merely a syntactic change. We had let clauses to create intermediate temporaries in the old syntax but not in the new syntax — you can't do `[for (x of y) let (z = ...) for (z of q) ...].]`