January 27, 2015 TC39 Meeting

The ES6 End Game

End-Game Schedule, Work back Part 1

- Early June 2015: Ecma GA approval at their semi-annual meeting
- April 1-June Ecma CC and GA review period, editorial preparation of publication document
- March 26, Approval of Final Draft at March 24-26 TC39 meeting (conditional assuming no last minute patent opt-outs will occur)
  - this is the last meeting date where TC39 can approve ES6 and achieve June GA approval
  - after this point, the only changes can be minor editorial or technical bug correction that don't require TC39 review

End-Game Schedule, Work back Part 2

- Feb. 20, Final Approval Draft Release
  - Member organizations need at least 30 days before voting to approve
  - Reported bugs will continue to be fixed
- Feb 2-19, Editor frantically incorporates Jan. meeting technical changes plus technical and editorial bug fixes to produce final candidate draft.
- Feb 1, 60 day patent opt-out review period starts
- Jan 27-29, TC39 meeting
  - Produce a small set of final technical changes for the editor to apply
  - This must be a *very small* delta from current spec. as there is really no time for major spec. change or for another technical review cycle
  - All changes carry the risk of introducing new bugs

Final ES6 RF Opt-out Period

- 60 day RF opt-out period for ECMA-262-6 and ECMA-402-2 starts February 1, 2015
  - End April 1, 2015
- Review documents:
  - ECMA-262 6th Edition, revision 31, TC39/2015/006
  - ECMA-402, 2nd Edition revision 6, TC39/2015/0004
  - Please factor into your review any technical decisions made that this meeting
- Note the end of the opt-out period falls after the March TC39 meeting. When the final approval vote at that meeting will need to be contingent upon no opt-outs arriving after the meeting.

ES6 Drafts Since Last Meeting

Rev 29, Dec. 6

- Support for normalizing relative module name
- RegExp changes discussed at Nov meeting
  - RexExp constructor no longer throws when the first argument is a RegExp and the second argument is present. Instead it creates a new RegExp using the same patterns as the first arguments and the flags supplied by the second argument.
  - Added 'flags' method to RegExp.prototype that returns a string containing the flag characters. This enables generic RegExp algorithms to pass through new flags added by RegExp subclasses.
  - Updated RegExp.prototype.split to work with RegExp subclass that over-ride exec to change the default matching algorithm.
  - Symbols @@match, @@replace, @@search, @@split, replace the corresponding string property names in RegExp prototype
  - Eliminate @@isRegExp and Symbol.isRegExp
- The this binding at the top level of a module has the value undefined.
- @@species defined on RegExp, Array, %TypedArray%, Map, Set, ArrayBuffer, Promise
- @@species pattern used to create derived instances of RegExp, Array, %TypeArray%, ArrayBuffer, and Promise, eliminates Zepto breakage
- Eliminated WeakMap/WeakSet clear methods
- String.prototype.includes replaces String.prototype.contains
- Within the spec, « » now used to bracket literal List values
- Eliminated deferred throw of first exception from Object.assign, Object.create, Object.defineProperties, Object.freeze, Object.seal, Object.isFrozen, Object.isSealed
- Template call sites objects are now canonicalized per realm, based upon their list of raw strings.
- Assigning to a const declared binding throw, even in non-strict mode code.
- Eliminated requirement to statically find and report assignment to const declared names.
- Made it a runtime error for a global let/const/class declaration to shadow a non-configurable property of global object.
- Eliminated unnecessary and observable (via strict get accessor) boxing before/in calls to GetIterator. This also makes ToObject unnecessary on RHS of destructuring assignment/binding
- Eliminate ToObject application to target argument of Reflect functions. Passing primitive values as targets now throw
- **`export default`** *ClassDeclaration* is now legal, includes allowing anonymous class declarations as default export

Rec 30, Dec 24

- Made it a runtime error to try to create a subclass of a generator function via a class declaration or class expression.
- Corrected bugs in String.prototype.match/replace/search/split in the logic for double dispatching to a RegExp agument
- Added language in 8.4 making it explicit that ES Jobs always run to completion
- Class name is in TDZ during extends clause evaluation
- Added static semantics that correctly validates statement label usage.
- Array.prototype.concat/push/splice throw TypeError if new length would be 2^53 or greater. (Bug 3409)
- Defined the Annex B (B.1.3) lexical grammar extensions for recognizing legacy HTML-like comments
- Added a forbidden extension item that forbids supporting HTML-line comments within modules.
- Additional early error for super() in non-constructor methods. (Already had an error for new super()).
- Object.prototype.toString now uses IsArray to when testing for built-in exotic array instances.
- IsArray abstract operation now longer throws if argument is a revoked proxy. It just returns false.
- Simplified and corrected early error naming rules for module exports
- Fixed several left-to-right evaluation order initialization bugs in various declaration forms

Rev 31, Jan 15.

- Merged AllocArrayBuffer and SetArrayBufferData into single abstract operation
- %TypedArray%.of now requires that its this value is a valid Typed Array constructor
- Replaced “moduleId” with “sourceCodeId” and eliminated most usage of such ids in module related abstract operations
- Some tweaking of Language Overview in 4.2
- A new document title

Plus major changes to implement Jan 7. teleconference instantiation reform concensus

New Document Title

Instantiation Reform: One last time

Instantiation Reform

- Problem:
  - Subclassing built-ins via [[CreateAction]] still has problems parameterizing allocation and has the potential to expose uninitialized instances
  - Continuing controversy over whether constructor parameters should be passed to [[CreateAction]]
  - Possibility of uninitialized instances must be explicitly dealt with in built-in constructors and all built-in methods that depend upon private state.
    - Would take a long time to implement
    - Not clear when implementations would actually provide subclassable built-ins.

Instantiation Reform Rev31

1. [[Construct]] takes a new additional argument *newTarget* which is the constructor object **`new`** was actually applied to. That argument also is also reified as arguments to the construct proxy trap and in **`Reflect.construct`**
2. Built-in object allocation and initialization are merged into a single constructor function, just like in ES5. Note that allocation take place in a base constructor rather than the original constructor and uses the *newTarget* argument to determine the [[Prototype]] of the new instance.
   - The subclass constructor determines what arguments are passed to the base constructor which may use those arguments in its allocation and initialization logic.
   - In addition to its use in determining the [[Prototype]], *newTarget* can be used to pass other information form a derived constructor to an allocating base constructor.
   - There is no [[CreateAction]] or similar separable allocation step.

Instantiation Reform Rev31

3. When a constructor is invoked via ordinary [[Construct]] and the constructor body was not defined using a class definition that has an **`extends`** clause, **`this`** is initialized to a newly allocated ordinary object whose [[Prototype]] is provided by the *newTarget* .
   - This is the way that function definition based constructors work today and must be maintained for legacy compatibility.
   - The same **`this`** initialization rules are used by both function definitions and class declarations that don't have extends clauses.
   - For the details covered by this proposal, an **`extends null`** clause is considered to be equivalent to an absent extends clause. *Issues*
4. When a constructor is invoked via ordinary [[Construct]], **`this`** is marked as uninitialized if the constructor body was defined using a class definition that has an **`extends`** clause.
   - If invoked via [[Call]], **`this`** is initialized in the normal manner.

Instantiation Reform Rev31

5. Any explicit reference to an uninitialized **`this`** throws a ReferenceError Exception
6. When **`this`** is in its uninitialized state, any expression in a constructor of the form **`super(`**<args>**`)`** accesses the [[Prototype]] of the active function and invokes [[Construct]] on it with the current *newTarget* value passed as the *newTarget* argument.
   - The value [[Construct]] returns sets the **`this`** binding.
   - Subsequent references to **`this`** produces the object value that was returned from the superclass constructor.
   - In other words, a super call delegates allocation and initial initialization steps to the super class constructor.
7. When **`this`** is in its initialized state, any expression of the form **`super(`**<args>**`)`** throws a ReferenceError exception. Rather than using [[Call]] to invoke the superclass constructor.

Instantiation Reform Rev31

8. When a function implicitly returns from a [[Construct]] invocation, if its **`this`** binding is still uninitialized a ReferenceError is thrown.
   - This covers the case where a constructor with an extends clause does not invokes **`super()`**.
9. When a function explicitly returns a non-object from a [[Construct]] invocation and its **`this`** binding is still uninitialized, a TypeError is thrown.
10. When a function explicitly returns a non-object from a [[Construct]] invocation but the **`this`** value has been initialized, the **`this`** value is returned as the value of [[Construct]]
    - This is required for ES1-5 compatibility

Instantiation Reform Rev31

11. If a class definition is for a derived class does not include an explicit constructor definition, it defaults to:
    ```
    constructor(...args)
    {super(...args)};
    ```

12. If a class definition is for a base class does not include an explicit constructor definition, it defaults to:
    ```
    constructor() {};
    ```
    – If you don't define a constructor body then you inherit both the constructor argument signature and body from your superclass.

Rev 31, Instantiations Reform Changes

- Added newTarget parameter to [[Construct]], Reflect.construct, and proxy 'construct' trap.
- Added NewTarget binding to Function Environment Records and abstract operations/methods of setting and accessing that binding
- Redefined [[Construct]] for ordinary ECMAScript functions so "derived" constructors don't preallocate the new object and bind this.
- Changed [[Construct]] for ordinary ECMAScript functions so "derived" constructors will throw if they try to explicitly return a non-object.
- Within constructors this binding has TDZ access semantics.
- super(...) syntactic form now illegal in Function/GeneratorDeclaration/Expression
- super(...) and new super(...) propagates current NewTarget to [[Construct]] call
- super(...) binds this value upon return from [[Construct]] call, throw if new already bound
- Refactored [[Call]] and [[Construct]] for ordinary ECMAScript functions so they can continue to shared common spec steps.
- Updated Generator object instantiation to work with new [[Construct]] design
- Refactored Function constructor and GeneratorFunction to share a common abstract operation based definition
- Every built-in constructor changed to merge object allocation and initialization code. In some cases (eg, TypedArray and Promise) significant refactoring of allocation and initialization logic.
- Updated [[Construct]] of Bound functions to handle newTarget parameter.

Rev 31 Instantiation Reform Open Issues

new super()

- **`new super()`** is a long standing feature of the spec.
  - It is syntactically a distinct special form
- Not discussed at Jan. 7 meeting
- In Rev 31: It is just like **`super()`** except it does not test or binding the current this value
  - Permits (in conjunction with return over-ride) writing a function definition that can be wired into a class hierarchy as a derived class

new super example

```
function Derived(...args){
    return
        new super(...process(args))
}
Derived.__proto__ = SuperClass;
Derived.prototype.__proto
  = SuperClass.prototype;
```

new.target syntax

- At the Jan 7 meeting a syntax was proposed that allows constructor code to access the *newTarget* value.
- Need if ES coded classes are expect to have the full power used by and needed to implement built-in constructors.
- Appeared to be consensus that this feature is needed
  - But perhaps could be deferred
- Generally positive reaction to proposed syntax but don't yet have consensus on inclusion.
- Marked with yellow hightlights in REV31 to indicate that it is only a proposed feature

8. ** Within a constructor, **new.target** can be used, as if it was an identifier, to access the *newTarget* argument.
   - new.target can be used to get information about the original constructor (such as its prototype property) that can be used to manually allocate the instance object. It can also be used to discriminate [[Construct]] vs. [[Call]] invocations.
   - An explicit return must be used to return such manually allocated objects

new.target syntax

- **`new.target`**
  - A new kind of *MemberExpression*.
  - A “meta-property”: *keyword* **`.`** *IdentifierName*
- Within a functon that is invoked via [[Construct]] it’s value is the current *newTarget* argument value.
- Within a [[Call]] invocation it’s value is **`null`** (**`undefined`** ??)

Final Issues

- Where is **super()** allowed