Making Built-in and Exotic Objects Subclassable

Allen Wirfs-Brock

The Basic Issue

- Object allocation and object initialization are separable issues.
- Subclassable abstractions requires program level control of this separation.
  - One allocation step
  - Followed by an initialization step for each level of subclassing.

Why Doesn’t This Work?

```
class Vector extends Array {
 constructor(...args) {super(...args)}
}
let a = new Array();
let v = new Vector();
a[5] = v[5] = 5;
console.log(a.length); //6 according to ES5
console.log(v.length); //0 according to ES5
```

Assuming straight forward desugaring of class definition and ES5 new operator semantics.

Because…

- Array uses a special exotic object representation that changes the semantics of [[DefineOwnObject]].
- 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

Why Doesn’t This Work?

```
class BetterDate extends Date{
 constructor(...args) {super(...args)}
}
let today = new BetterDate(2013,0,26);
console.log(today.getMonth());
    //Throws TypeError according to ES5
```

Assuming straight forward desugaring of class definition and ES5 new semantics.

Because...

- **new Date** creates an object with [[Class]]== “Date”.
- The object that **new BetterDate** creates has [[Class]]== “Object”.
- The **getMonth** method throws when this.[[Class]] != “Date”
- But going deeper...
  - Implementations use a special object layout for Date instances with a fixed position timeValue slot.
  - Built-in date methods directly access the this object’s timeValue slot
  - The [[Class]]==“Date” check ensures that the methods don’t try to access that slot in objects where it doesn’t exist.

Deeper Why – [[Construct]]

- Built-in constructors like Array and Date essentially have special [[Construct]] internal methods that know how to create their special flavor of objects.
- Internal method implementations are not inherited.
- All user defined objects use the standard [[Construct]] that always allocates an ordinary object:
  1. Let *obj* be a new ordinary object with all the standard internal methods.
  2. Set *obj's* [[Class]] to "Object"
  3. Let *proto* be [[Get]] of this constructors "prototype" property
  4. Set *obj's* [[Prototype]] to *proto*.
  5. Call this constructor with *obj* as the this value and the original argument list
  6. Return *obj*.
- Currently, in ES6 the only way to define an alternative [[Construct]] is via a Proxy

First Try at a Fix

- Use pretty much normal [[Construct]] for built-ins.
- Move magic initialization (internal data properties and internal methods) into the constructor function, post object allocation.
- Internal data properties need to be expandos (probably based upon private symbols)
- Built-in methods use internal data property sniffing instead of [[Class]] brand check.
- Now this would work:

```
class BetterDate extends Date{
  constructor(...args) {super(...args)}
}
let today = new BetterDate(2013,0,26);
console.log(today.getMonth());   // 0 – it now works...
```

But...

Issues with First Try

- Ollie’s objection:
  - Doesn’t really want internal data properties to be expandos.
  - Implementers want to allocate different machine/C level data structures for different kinds of built-ins.
- Allen’s objection:
  - What about internal method conflicts?
- Jason’s objection:
  - More than one magic constructor can be applied to an object:

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

It’s both a Date and a Map! Oh my!

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 allocation 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*.
- 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]())`

Some Examples

- Built-in Array[@@create]
  - Allocates an exotic Array object
  - Installs non-configurable “length” property
- Built-in Date[@@create]
  - Allocates an ordinary object
  - Associates a [[DateValue]] internal data property with the object to hold time value.
  - But it could transparently be an special implementation record structure

Subclasses inherit @@create from superclass constructor

```
class Vector extends Array {}
let v = new Vector();
v[5] = 5;
console.log(v.length); //6
```

- Because new Vector uses @@create inherited from Array which creates an exotic array object with magic length behavior.

```
class BetterDate extends Date {}
let today = new BetterDate(2013,0,26);
console.log(today.getMonth());
```

- Because new BetterDate uses @@create inherited from Date which creates an implementation dependent object structure with the internal data property needed by getMonth.

Built-ins with @@create Methods

- Array, String, Boolean, Number, Date, RegExp, Map, Set, Weakmap, ArrayBuffer, *TypedArray*, etc.
  - Pretty much everything that is internally branded or might have an implementation dependent representation.
- Built-in @@create methods are non-writable, non-configurable.
  - Just like built-in "prototype" properties
  - Both are critical to the integrity of the built-in constructors.

@@create Also Useful for Application Classes

- DIY branding:

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

   static [$$create] () {
     let obj = super();
     Object.defineProperty(obj,$fooBrand, {value: true});
     return obj;
   }
};
```

- Creating Proxy based instances:

```
class EmulatedArray extends Array.prototype{
   static [$$create] () {
     let target= Object.create(this.prototype,{length:{value:0, writable:true}};
     let obj = Proxy(target,{defineProperty(obj,p,desc) {
          ... //logic to emulate standard Array defineProperty behavior
     }});
     return obj;
   }
};

let e = new EmulatedArray();
e[5] = 5;
console.log(e.length); //6
Console.log(Array.isArray(e));  //false!
```

In the future, @@ might be auto generated to allocated instance properties explicitly declared via syntax in a class definition.

ES5 Built-in Branding

Consider this ES5 + reality code:

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

var t= new Thing;
 t[5] = 5;
console.log(t.length); //6
console.log(Array.isArray(t)); //true
console.log(t.toString());   //[object Array]
```

Built-in branding is based upon 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]

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

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

```
let ex = new class extends RegExp{ };
console.log({ }.toString.call(ex)); //[object RegExp]
```

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 “an RegExp instance”
- In ES6 spec. all [[Class]] uses can be replaced with language like:
  - “is an exotic array object” ⬄ [[Class]]==“Array”
  - “has a [[Match]] internal data property ⬄ [[Class]] ==“RegExp”
  - Etc.
- Like always, it’s left to implementations to decided how such tests actually work

When does a Constructor need to act as a Constructor?

- `Foo()` vs. `new Foo()` vs. `super()` call of Foo
  - Built-ins historically rely on [[Call]] vs. [[Constructor]] distinction.
  - ECMAScript code doesn't have any way to make that distinction
  - super() constructor calls work like "called as a function" but usually want the behavior of "called as a constructor"

Testing the this value almost works

```
function Foo() {
   if (this === undefined) return new Foo();
   this.state = “initialized”;
 }
let f1 = Foo(); //“called as a function”, this is undefined ☺
let f2 = new Foo(); //”called as a constructor”, this in an object ☺

class SubFoo extends Foo{constructor(){super()}};
let f1 = new SubFoo(); //Foo “called as a function”, this is an object ☺

let namespace = {Foo: Foo};
let f3 = namespace.Foo();
            //Foo “called as a function”, this is an object ☹
           // namespace object initialized as a Foo instance
```

Testing the this value for undefined isn’t the answer

- Not:

```
function Foo() {
  if (this === undefined) return new Foo();
  this.state = “initialized”;
 }
```

- Instead:

```
function Foo() {
  if (Type(this)!== “Object” ||
      this.state == “uninitialized” ) return new Foo();
  this.state = “initialized”;
 }
```

A spec. language cheat

Constructors need to be able to recognize uninitialized instances

- Built-ins can do this via existing internal properties
  - RegExp @@create: Set [[Match]] internal property to undefined to indicate uninitialized
  - RegExp constructor: Setting [[Match]] internal property to pattern indicates initialized
- ECMAScript code can define their own flags

```
const $fooBrand = Symbol(true);
export class Foo {
   constructor() {
      if (typeof this !== “object” || !this[$fooBrand]} return /*CAAF case */
      /* constructor initialization case */
      Object.defineOwnProperty(obj,$fooBrand, {value: true, writable: false});
   }

   static [$$create] () {
     let obj = super(create);
     Object.defineOwnProperty(obj,$fooBrand, {value: false, writable: true});
     return obj;
   }
};
```

Constructor functions initialize uninitialized instances

@@create functions mark new instances as uninitialized

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

- Add one bit of state to every object: initialized/ uninitialized.
- Built-in @@create methods set new object state to uninitialized.
- Object.call(uninitObj) and other built-in constructors set uninitialized 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 uninitialized state.
- Object.create, { }, [ ], and various built-in functions create objects that are all ready initialized (backwards compat)

Jan 29 TC39 meeting decision: defer

Examples with object model init state

```
class Foo { }
let f = new Foo();
          // f is initialized because of implicit super.constructor call
        //  to Object constructor which marks this obj as initialized

class Bar {
    constructor () {
        this.prop = 42;  //initialize some state
        super();   //marks object as initialized (could do it first)
     }
}
let b = new Bar();

class Baz {
    constructor () {
        if (Object.isInitialized(this))
              return new Baz();  //called as function case
        super();  //called as constructor or super.constructor case
     }
}
let bz1 = new Baz();
let bz2 = Baz();
```

Jan 29 TC39 meeting decision: defer

Various Oddities and Backward Issues and how to fix them - 1

- Existing code inherits from Array.prototype and doesn't expect subclass behavior.
  - General solution, Array[@@create] is what marks an object as an array/array-subclass object
    - But user defined @@create methods also can, via a @@symbol
  - Object.create(Array.prototype)//not a subclass
  - new class extends Array{ } // is a subclass

Various Oddities/Backward Compat Issues and how to fix them - 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
      - It may make sense to parameterize result class (like Smalltalk species)
  - 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(?)

Various Oddities/Backward Compat Issues and how to fix them - 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 instances.
  - String methods delegate to RegExp methods.