Unintitialized Objects

Brendan: “The most important thing here (I agree with Andreas R.) is -- if possible -- avoiding uninitialized object observability.”
http://esdiscuss.org/topic/new#content-49

Can we define what it mean for an object to be “initialized”?

- An object can represent an abstraction of arbitrary complexity
  - An abstraction isn’t just its “root object” but also all the constituents objects and values that make up the encapsulated state of the abstraction
  - All the invariants that relate to all those constituent parts.
- So, deciding whether an object has been initialized can be arbitrarily complex.

The concept of a correctly initialized object lies in the application domain, not in the language design domain.

- We can't guarantee completion of application level object initialization

- We can (and do) guarantee any initialization necessary for system-level runtime integrity
  - For ES objects, this is just the essential runtime invariants
  - Host supplied objects, need to understand the differences between their application-level initialization and initialization necessary for system-level integrity.

Another concern

“Hideous Number special-casing spread around in the draft”

- What’s this talking about:

  If Type(*value)* is Object
  and *value* has a [[NumberData]] internal slot
  and the value of *value’s* [[NumberData]] internal slot
  is **undefined**,
  then throw a **TypeError** exception.

- This specific check actually occurs exactly one place in the spec:
  http://people.mozilla.org/~jorendorff/es6-draft.html#sec-properties-of-the-number-prototype-object
- What’s it doing: checking if we are trying to access the value of an unitialized Number object.

But...

- But similar patterns also occurs for other built-in classes.
- For some built-in classes, they occur at more than one place
  - Because of the need to propagate exceptions, abstracting the test doesn't save much
- But most occurrences are only within the section of the spec that is defining the related build-in class.

Why is this pattern needed?

1. Desire to separate object allocation from initialization.
2. Legacy built-ins that have different, “called as constructor”, “called as function” behavior.
3. Necessitating need to distinguish initialized and uninitialized instances.
4. But exacerbated by spec. level decision to use some internal slots as both initialization flags and value holders.

Current ES6 Spec:

- The @@create method of an object *F* performs the following steps:
  1. Let *F* be the **this** value.
  2. Let *obj* be OrdinaryCreateFromConstructor(*F*, **"%NumberPrototype%"**, ([[NumberData]])).
  3. Return *obj*.

Undefined means uninitialized

- It could just as easily be:
  1. Let *F* be the **this** value.
  2. Let *obj* be OrdinaryCreateFromConstructor(*F*, **"%NumberPrototype%"**, ([[NumberData]], [[NumberInitialized]])).
  3. Set *obj*'s [[NumberData]] internal slot to **NaN**.
  4. Return *obj*.

Undefined means uninitialized

This sort of optimization probably should be in the spec, but there was resistance to adding more "internal properties to legacy built-ins.

In theory, not needed at all for most new built-in classes

- Map[Symbol.create]
  1. Let *F* be the **this** value.
  2. Let *obj* be the result of calling OrdinaryCreateFromConstructor(*F*, **"%MapPrototype%"**, ([[MapData]])).
  3. Let obj's [[MapData]] internal slot be an empty List.
  4. Return *obj*.
- Then this would be legal:

```
let m = Map[Symbol.create]();
m.set("foo",  "bar");
```

This line not in the current ES6 spec. draft.

No test needed to see if map instance has been initialized

But would you be ok with...

```
let m = Map[Symbol.create]();
Map.call(m, somethingWithEntries);
Map.call(m, somethingElseWithEntries);
//m has entries from both,
//or do you want Map to implicitly do a clear
//or should Map throw if m isn't empty?
```