Ecma/TC39/2012/053

Direct Proxies: open issues

Tom Van Cutsem (VUB)

getPrototypeOf trap

- Presuming __proto__ is specified in Annex B => writable __proto__ already destroys the invariant that the [[Prototype]] link is stable.

```
var p = Proxy(target, handler);
Object.getPrototypeOf(p) // => handler.getPrototypeOf(target)
```

- However, frozen objects should continue to have stable prototype-chain
- getPrototypeOf trap result should be consistent with target object’s proto

getPrototypeOf trap

- How to spec interceptable [[Prototype]]?
  - [[Prototype]] is currently an internal property
  - Would need to become an internal “accessor” property or split up into [[GetProto]] / [[SetProto]] methods
  - [[GetProto]] / [[SetProto]] would trigger traps for proxies

__proto__ & get/set traps

- Interaction between magical __proto__ and proxies:
- proposal: proxy.__proto__ should just trigger the proxy’s get trap.
- Handler gets to decide whether this property name is magical or not.

```
var p = Proxy(target, handler);
p.__proto__     // => handler.get(target, “__proto__”, p)
p.__proto__ = x // => handler.set(target, “__proto__”, x, p)
```

Trapping instanceof

- Use cases for extending Function [[HasInstance]] behavior.
- Point in case: x instanceof Global answering true even if x and Global live in separate frames/windows
- Original proposal:

```
var fp = Proxy(targetFunction, handler);
x instanceof fp // => handler.hasInstance(targetFunction, x)
```

- Note: fp gets access to x. Is this problematic? ([[HasInstance]] already specified this way internally)

Trapping Object.isExtensible etc.

- Currently, Object.isExtensible doesn't trap (same for isSealed, isFrozen):

```
var p = Proxy(target, handler);
Object.isExtensible(p) // => Object.isExtensible(target)
```

- Makes it impossible for membranes to accurately report extensibility across a membrane:

```
// shadowTarget holds wrapped non-config props of realTarget
var membraneP = Proxy(shadowTarget, handler);
Object.isExtensible(shadowTarget) // true
Object.isExtensible(realTarget)   // true
Object.isExtensible(membraneP)    // true

Object.preventExtensions(realTarget)

Object.isExtensible(realTarget)   // false
Object.isExtensible(shadowTarget) // still true!
```

Trapping Object.isExtensible etc.

- Proposal:

```
var p = Proxy(target, handler);
Object.isExtensible(p) // => handler.isExtensible(target)
```

- Same for isSealed, isFrozen

- With assertions so that the trap cannot “lie”: if target is non-extensible, isExtensible trap cannot return true (and the other way around)

- Problem solved for membranes:

```
isExtensible: function(shadowTarget) {
  ...
  return Object.isExtensible(realTarget);
}
```

Direct Proxies: “internal” properties

- Direct proxies wrapping built-ins, e.g. Date instances
- Current proposal is to auto-unwrap internal properties such as [[PrimitiveValue]], i.e.

```
Proxy(aDate, aHandler).[[PrimitiveValue]]
=> aDate.[[PrimitiveValue]]
```

- Primitive methods on Date.prototype should work fine on proxies for Dates:

```
var d = new Date();
var p = Proxy(d, handler);
Date.prototype.getTime.call(p);
// => Date.prototype.getTime.call(d)
```

Direct Proxies: “internal” properties

- Issue raised by Jason Orendorff: auto-unwrapping is dangerous if built-in methods return non-primitive values (e.g. object references)

- Point in case: ES6 iterators next() method

```
var arr = [obj0, obj1, obj2];
var it = arr.iterator();

var membraneP = wrap(it);

Iterator.prototype.next.call(membraneP)
// if we auto-unwrap => membraneP leaks obj0
// could also be non-transparent and throw TypeError
```

Proposal: nativeCall trap

- Instead of auto-unwrapping, delegate to a generic trap (which auto-unwraps by default):

```
var d = new Date()
var p = Proxy(d, handler);
Date.prototype.getTime.call(p)
// => handler.nativeCall(d, Date.prototype.getTime, [])
// defaults to Date.prototype.getTime.call(d)
```

```
var it = array.iterator()
var p = Proxy(it, handler);
Iterator.prototype.next.call(p)
// => handler.nativeCall(it, Iterator.prototype.next, [])
// => membrane can wrap result
```

Proposal: nativeCall trap

- Which non-generic built-in methods would trigger this trap?
- Non-generic methods defined on {Boolean,Date,Number,RegExp}.prototype that check the type of `this`
- String.prototype methods mostly try to coerce `this` ToString so don't require this mechanism.
- Host object methods?

defaultValue

- add a defaultValue trap?
- Nov. 2011 meeting -> more in favor of exposing it via a private name (enables custom behavior for non-proxy objects as well)

```
var toString = Object.prototype.toString;
var valueOf  = Object.prototype.valueOf;
var p = Proxy(t, handler)

toString.call(p) // => handler.defaultValue(t, “string”)
valueOf.call(p)  // => handler.defaultValue(t, “number”)

toString.call(p)// => handler.toString(t)
valueOf.call(p) // => handler.toNumber(t)

toString.call(p)// => p[defaultValueName](“string”)
valueOf.call(p) // => p[defaultValueName](“number”)
```

Proxies & private names

- Proposal:
  - `proxy[privateName]` should not trigger the `get` trap (so the property name argument to the `get` trap can remain a simple string)
  - a separate `getName` trap
- `proxy[name] => handler.getName(target, name.public)` (still no private name leakage to handler)

Proxies & private names

- `getName(target, name.public)` trap should return:
    - a pair `[name, value]`, proving to the proxy that the handler really knows about the private name
    - `undefined`, signal for: “I don’t know about this name property, please forward to target”
- if handler doesn’t implement `getName` trap, default is to forward to target

VirtualHandler

- VirtualHandler fundamental traps currently throw (abstract methods)
- Propose to have these forward to the target instead
- rename `VirtualHandler` to just `Handler`?
    - any subclass of Handler can still choose to ignore the target object in its fundamental traps

```
var h = new Handler();
h.defineProperty = function(){...};
h.deleteProperty = function(){...};
var p = Proxy(target, h);
```

Freeze, seal, defineOwnProperties

- Came up when specifying “derived traps” in Handler.prototype
- Specify best-effort semantics?