Ecma/TC39/2011/053

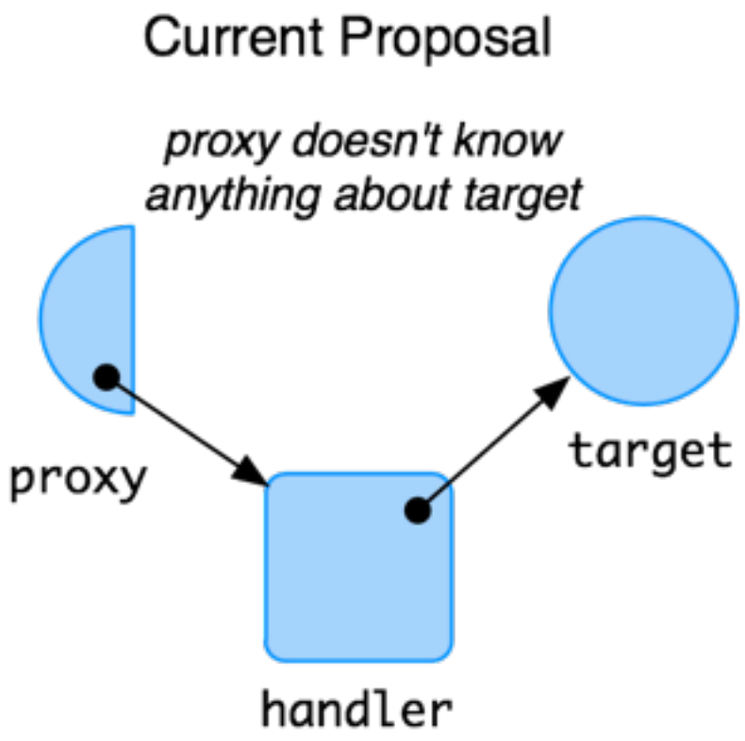
Harmony Proxies: update

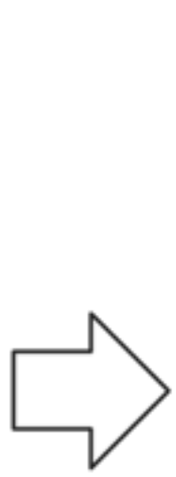
Tom Van Cutsem (Vrije Universiteit Brussel)
Mark S. Miller (Google)

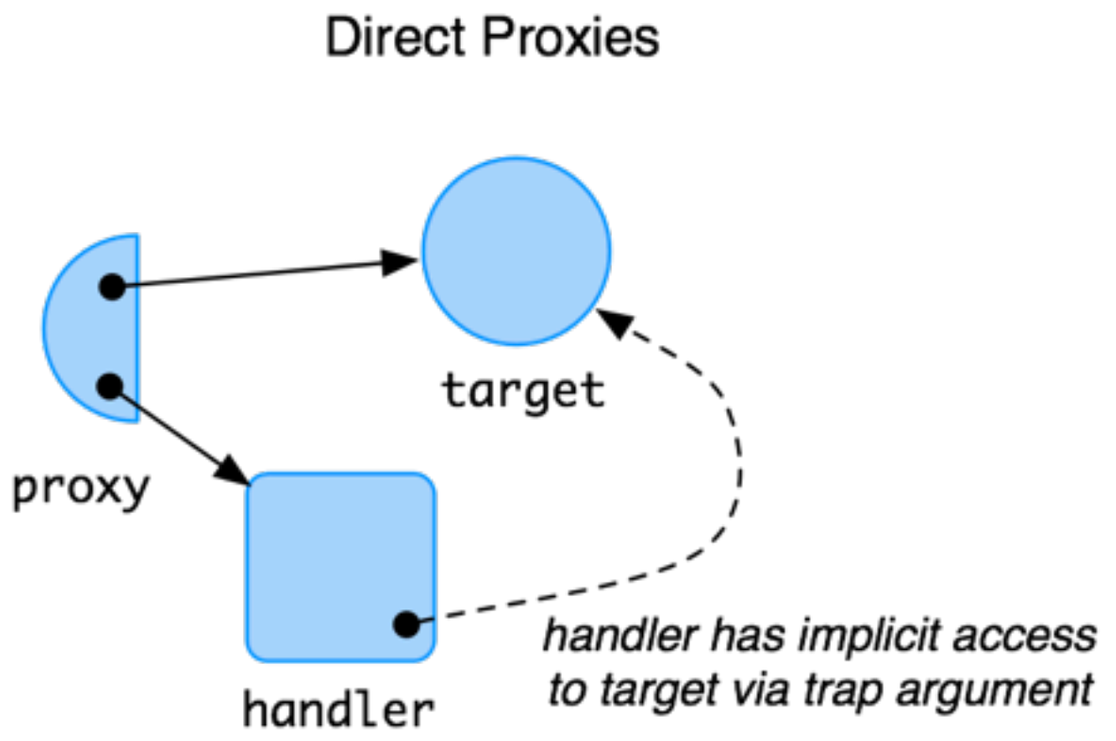
Direct Proxies: motivation

- Enable proxies to emulate non-configurable properties
- Enable proxies to emulate non-extensible, sealed and frozen objects
- Do so without violating non-configurability and non-extensibility invariants
- Enable proxies with a custom [[Class]], typeof, and enable sensible wrapping of host objects
- Bonus: unify object and function proxies

Direct Proxies








```
var handler = new ForwardingHandler(target)
var proxy = Proxy.create(handler,
  Object.getPrototypeOf(target))
```

```
var proxy = Proxy.for(target, handler)
```

Direct Proxies: “internal” properties

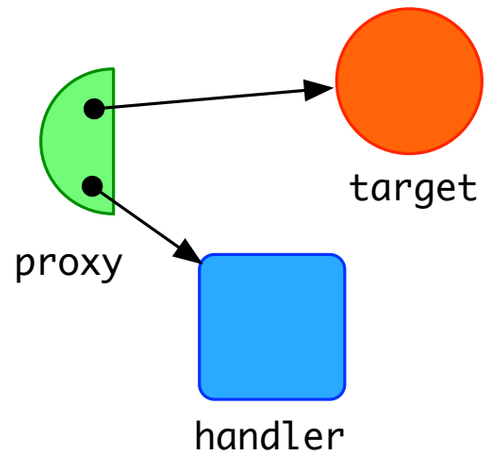
- Direct proxy inherits properties such as [[Class]] from its target object

```
var d = new Date();
var p = Proxy.for(d, {});

Object.prototype.toString.call(p) // "[object Date]"
Object.getPrototypeOf(p) // Date.prototype
typeof p // "object"
p === d // false
```

API Changes

- All traps optional
- Missing trap: default to forwarding the operation to the wrapped target
- No more fundamental vs. derived traps

Direct Proxies: updated handler API




```
Object.getOwnPropertyDescriptor(proxy,name)
~~Object.getPropertyDescriptor(proxy,name)~~
Object.defineProperty(proxy,name,pd)
Object.getOwnPropertyNames(proxy)
~~Object.getPropertyNames(proxy)~~
delete proxy.name
for (name in proxy) { ... }
Object.{freeze|seal|preventExtensions}(proxy)
name in proxy
({}).hasOwnProperty.call(proxy, name)
Object.keys(proxy)
proxy.name
proxy.name = val
proxy(...args)
new proxy(...args)
```

```
handler.getOwnPropertyDescriptor(name,target)
~~handler.getPropertyDescriptor(name,target)~~
handler.defineProperty(name,pd,target)
handler.getOwnPropertyNames(target)
~~handler.getPropertyNames(target)~~
handler.delete(name,target)
handler.enumerate(target)
handler.protect(operation,target)
handler.has(name,target)
handler.hasOwn(name,target)
handler.keys(target)
handler.get(receiver,name,target)
handler.set(receiver,name,value,target)
handler.call(receiver,args,target)
handler.new(args,target)
```

Fix trap => protect trap

- Fix trap renamed to protect trap (proxies are no longer “fixed”):

```
var p = Proxy.for(target, handler);
Object.preventExtensions(p);
  // calls handler.protect(‘preventExtensions’)
Object.seal(p);   // calls handler.protect(‘seal’)
Object.freeze(p); // calls handler.protect(‘freeze’)

// p remains trapping!
```

Unifying Object and Function proxies

- If typeof target === “function”, “call” and “new” traps are enabled

```
var p = Proxy.for(aFunction, handler);

p(1,2,3);     // handler.call(undefined, [1,2,3], aFunction)
new p(1,2,3); // handler.new([1,2,3], aFunction)

Function.prototype.call.call(p,rcvr,1,2,3)
  // handler.call(rcvr, [1,2,3], aFunction)

Object.prototype.toString.call(p);   // “[object Function]”
Function.prototype.toString.call(p); // code of aFunction
```

Non-configurability & non-extensibility invariants

- A direct proxy enforces non-configurability & non-extensibility invariants
  - Check if intercepted property on the target is non-configurable
  - Check if the target is non-extensible
- If invariant violation detected => throws TypeError

Non-configurability: invariants

sealed property = a non-configurable own property of the target
frozen property = a non-configurable non-writable own property of the target

`get{Own}PropertyDescriptor`	must return compatible descriptors for existing sealed properties, cannot return non-configurable descriptors for configurable or non-existent properties
`defineProperty`	must return false for incompatible changes made to sealed properties
`delete`	must return false for sealed properties
`has{Own}`	must return true for sealed properties
`get`	must return [[SameValue]] values for frozen data properties, must return undefined for sealed accessors with an undefined getter
`set`	must return false for frozen data properties unless new value is [[SameValue]], must return false for sealed accessors with an undefined setter

Non-configurability: invariants

Cost of these checks is dependent on size of the target

`get{Own}PropertyNames`	must report all sealed properties
`keys, enumerate`	must report all enumerable sealed properties

Non-extensibility: invariants

new property = a property that does not exist on a non-extensible target

`getOwnPropertyDescriptor`	must return undefined for new properties
`defineProperty`	must return false for new properties
`getOwnPropertyNames, keys`	must not list new property names
`hasOwn`	must return false for new properties

Proxies that don’t (need to) wrap a target?

- Still possible to create fully “virtual” proxies: just create a direct proxy with a fresh (empty) target object.

```
// assuming handler has no missing traps
Proxy.create = function(handler, proto) {
  return Proxy.for(Object.create(proto), handler);
};
```

- We could allow target to be `null`. Proxy would then throw when exposing non-configurable properties or when protected (cf. restrictions of the current design)

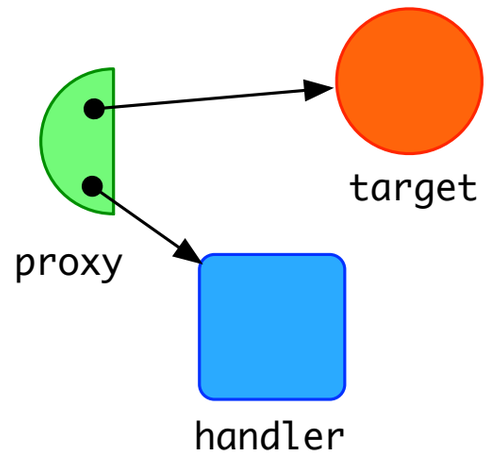
Reflect module

- Reflect module replaces “default forwarding handler”
- Enables easy manual forwarding of operations. Example:

```
funtion createChangeObserver(target) {
  return Proxy.for(target, {
    set: function(receiver, name, value, target) {
      print('property '+name+' on '+target+' set to '+value);
      return Reflect.set(target, receiver, name, value);
    }
  });
}
```

- Dual of proxy traps. Enables generic forwarding of traps (double lifting)

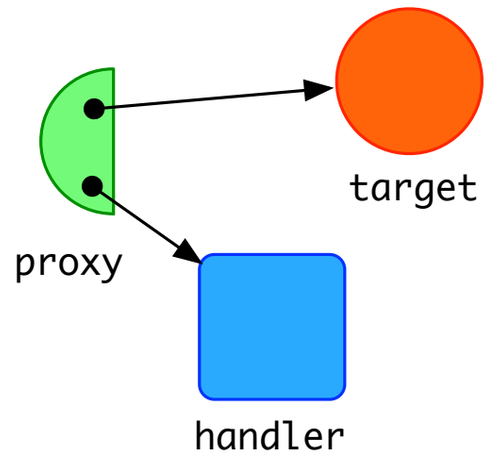
Reflect module




```
Object.getOwnPropertyDescriptor(proxy,name)
Object.defineProperty(proxy,name,pd)
Object.getOwnPropertyNames(proxy)
delete proxy.name
for (name in proxy) { ... }
Object.{freeze|seal|preventExtensions}(proxy)
name in proxy
({}).hasOwnProperty.call(proxy, name)
Object.keys(proxy)
proxy.name
proxy.name = val
proxy(...args)
new proxy(...args)
```

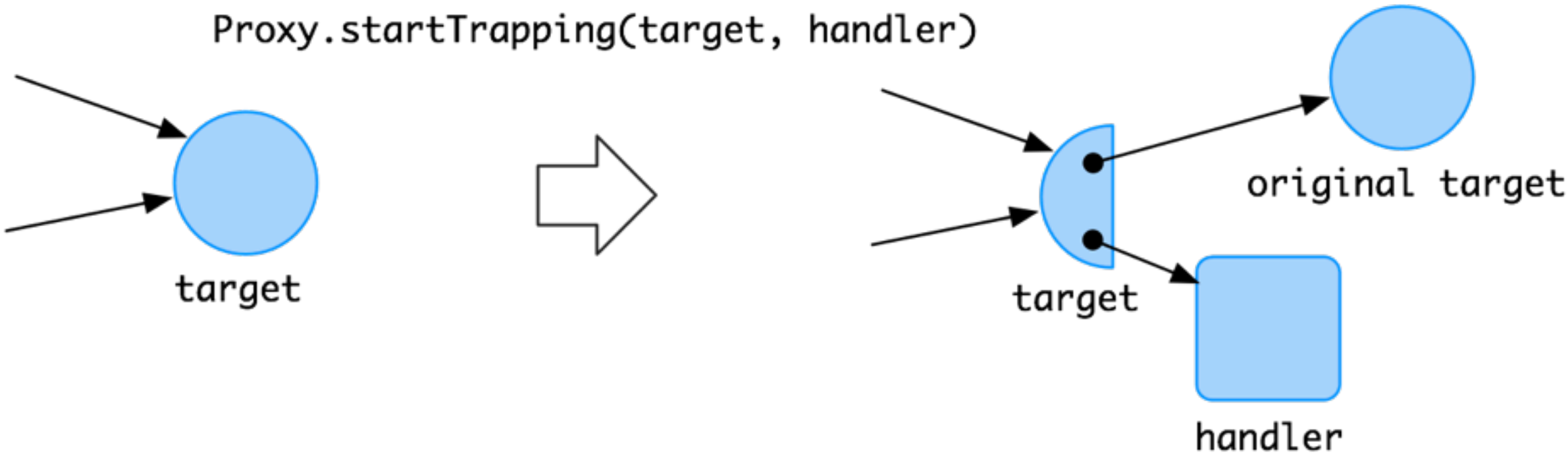
```
handler.getOwnPropertyDescriptor(name,target)
handler.defineProperty(name,pd,target)
handler.getOwnPropertyNames(target)
handler.delete(name,target)
handler.enumerate(target)
handler.protect(operation,target)
handler.has(name,target)
handler.hasOwn(name,target)
handler.keys(target)
handler.get(receiver,name,target)
handler.set(receiver,name,value,target)
handler.call(receiver,args,target)
handler.new(args,target)
```

Reflect module




```
handler.getOwnPropertyDescriptor(name,target)   Reflect.getOwnPropertyDescriptor(target,name)
       handler.defineProperty(name,pd,target)   Reflect.defineProperty(target,name,pd)
          handler.getOwnPropertyNames(target)   Reflect.getOwnPropertyNames(target)
                 handler.delete(name,target)    Reflect.delete(target,name)
                   handler.enumerate(target)    Reflect.enumerate(target)
            handler.protect(operation,target)   Reflect.protect(target,operation)
                    handler.has(name,target)    Reflect.has(target,name)
                 handler.hasOwn(name,target)    Reflect.hasOwn(target,name)
                       handler.keys(target)     Reflect.keys(target)
            handler.get(receiver,name,target)   Reflect.get(target,receiver,name)
     handler.set(receiver,name,value,target)    Reflect.set(target,receiver,name,value)
          handler.call(receiver,args,target)    Reflect.call(target,receiver,args)
                 handler.new(args,target)       Reflect.new(target,args)
```

startTrapping (a.k.a. Proxy.attach)




```
let t = {};
let alias = t;
t === alias; // true
let origT = Proxy.startTrapping(t, h);
t === alias; // true
origT === t || origT === alias; // false
```

startTrapping: concerns

- Objects should be able to protect themselves against attachment
  - Proposal: align such protection with non-extensibility

```
let t = Object.freeze({});
Proxy.startTrapping(t, h); // TypeError
```

  - invoking startTrapping on a proxy should trigger a trap to allow the existing handler to reject

- Weakmaps: existing bindings keyed by target should remain unaffected

```
let t = {}, wm = new WeakMap();
wm.set(t, 42);
wm.get(t); // 42
let origT = Proxy.startTrapping(t, h);
wm.get(t); // 42
wm.get(origT); // undefined
```

Direct Proxies: stopTrapping

- A stopped proxy behaves as a fully transparent forwarder to target (only distinguishable by identity)


```
var proxy = Proxy.for(target, handler)
```

```
// Better API:
{proxy, stopTrapping} = Proxy.revokable(target,handler);
// calling stopTrapping() “fixes” the proxy, drops the handler
```

Refactoring prototype climbing in the specification

- Problem: interaction between proxies and inherited property access/ assignment/lookup is non-intuitive and requires deep knowledge of the spec.

```
var p = Proxy.for(target, handler);
p[name]; // triggers handler.get(p,name,target)
var child = Object.create(p);
child[name]; // triggers handler.getPropertyDescriptor(name)
```

Refactoring prototype climbing in the specification

- Solution: refactor [[Get]], [[Put]] and [[HasProperty]] for Objects such that these algorithms *themselves* walk the prototype chain, rather than delegating to [[GetProperty]]

```
var p = Proxy.for(target, handler);
p[name]; // triggers handler.get(p,name,target)
var child = Object.create(p);
child[name]; // triggers handler.get(child,name,target)
```

Refactoring: changes to the Proxy API

- Get rid of Object.getPropertyDescriptor and Object.getPropertyNames
- Get rid of the corresponding traps for proxies
- get and set traps again take a “receiver” parameter
- “has” trap now also called for proxies-as-prototypes
- add Reflect.get, Reflect.set built-ins

Reflect.get, Reflect.set

- Problem: once a [[Get]]/[[Put]] request on a child has been intercepted by as proxy-used-as-prototype, how to transparently forward/continue the operation?

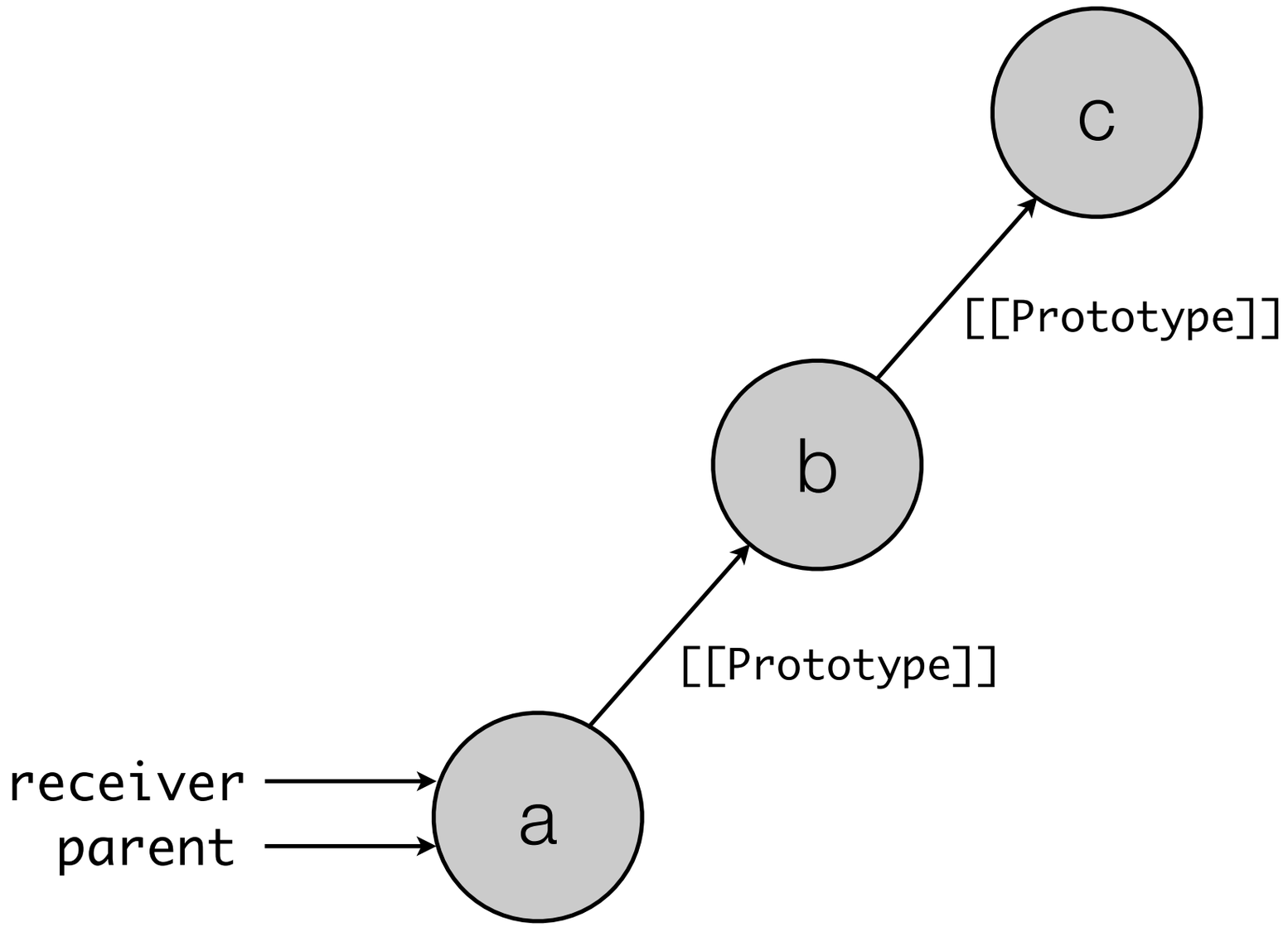
```
set: function(receiver, name, value, target) {
  // Wrong! If name is an accessor, will bind |this|
  // to |target| instead of to |receiver|
  target[name] = value;
  return true;
}
```

Reflect.get, Reflect.set

- Solution: introduce built-ins that expose the new prototype-climbing variations of [[Get]] and [[Put]], including the ability to differentiate the “current parent” from the “initial receiver”:

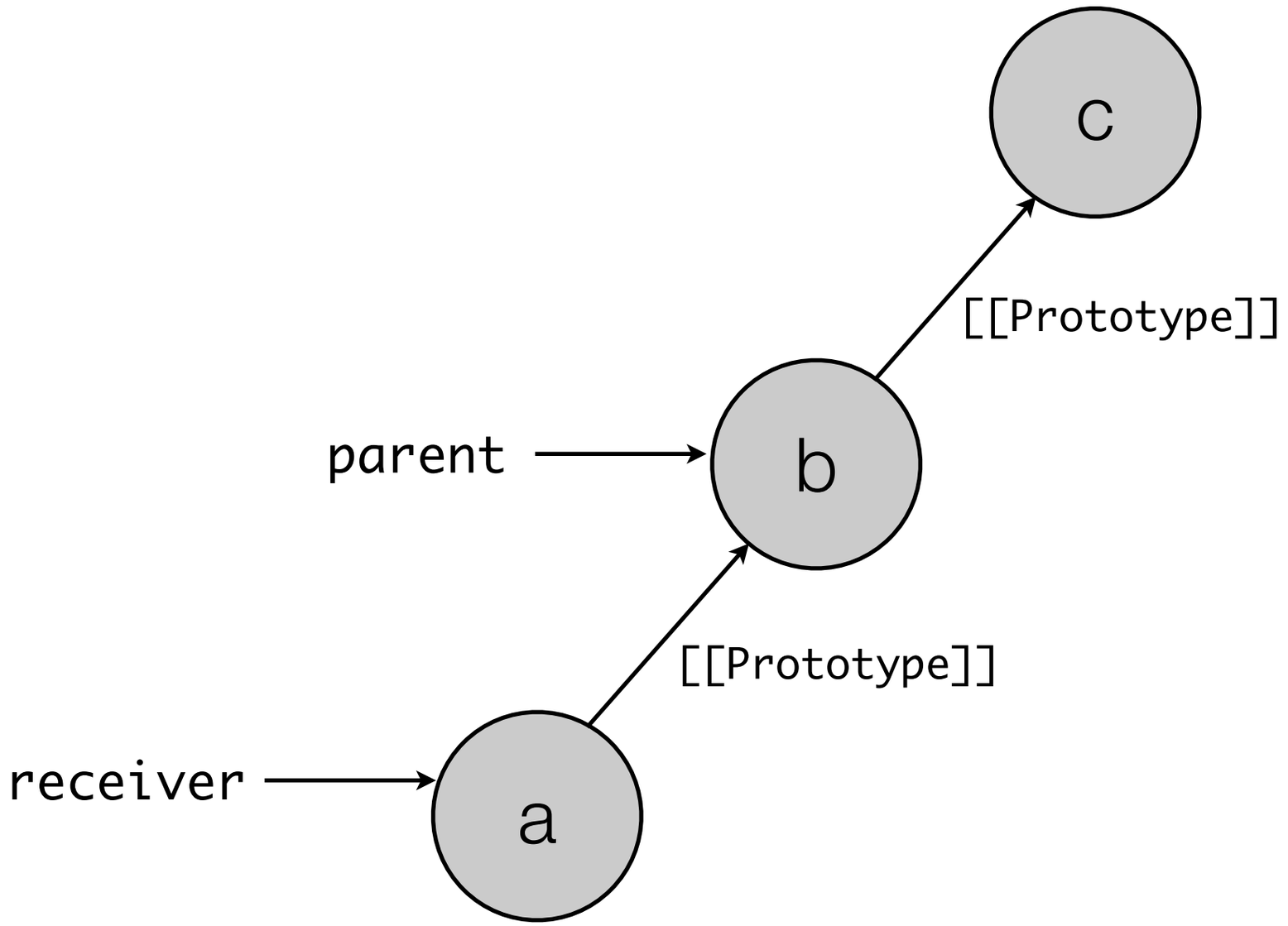
```
set: function(receiver, name, value, target) {
  return Reflect.set(target, receiver, name, value);
}

// Reflect.set(parent,receiver,name,value) -> boolean

// receiver[name] = value
// is equivalent (modulo return value) to
// Reflect.set(receiver, receiver, name, value)
```

parent vs receiver



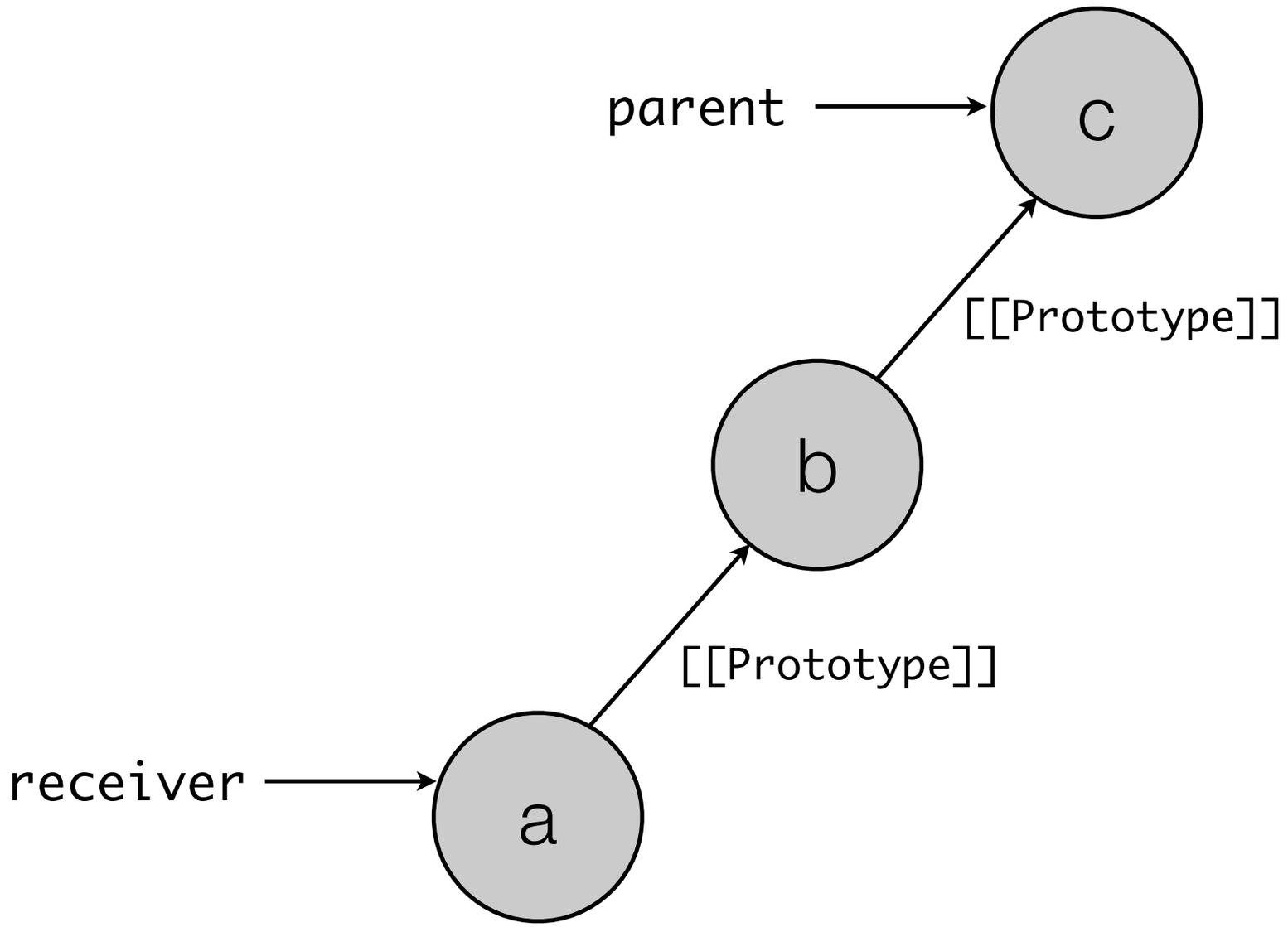

```
a['x'] = 0;
Reflect.set(a,a,'x',0);
```

parent vs receiver



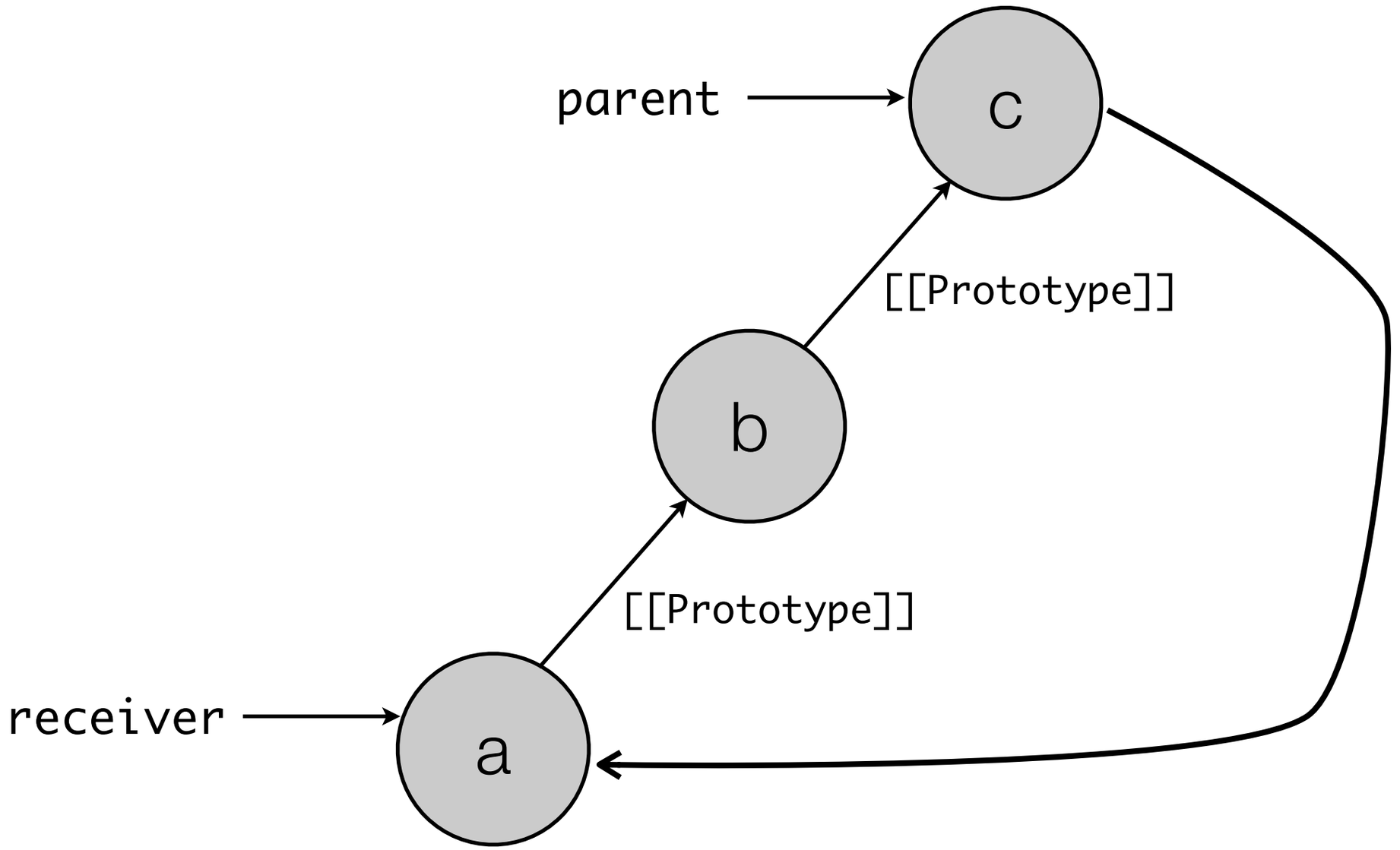

```
a['x'] = 0;
Reflect.set(b,a,'x',0);
```

parent vs receiver




```
a['x'] = 0;
Reflect.set(c,a,'x',0);
```

parent vs receiver




```
a['x'] = 0;
Object.defineProperty(a,'x',{...});
```

Other pending proxy strawmen

- defineProperty reject behavior (return boolean): consensus?
- Superseded by direct proxies:
    - Function proxy prototypes
    - Derived getPropertyDescriptor/getPropertyNames traps
    - Handler access to proxy
    - Derived trap implementation of default forwarding handler

Other pending proxy strawmen

- Superseded by refactoring prototype climbing in the spec:
  - proxy set trap (avoid redundant trap invocation)
  - proxy drop receiver
  - open issue in default forwarding handler: how to express default “set” trap

Misc

- add a defaultValue trap?
    - enables stratified toString / valueOf support
    - required to faithfully emulate Date semantics?