Virtualizing the Object

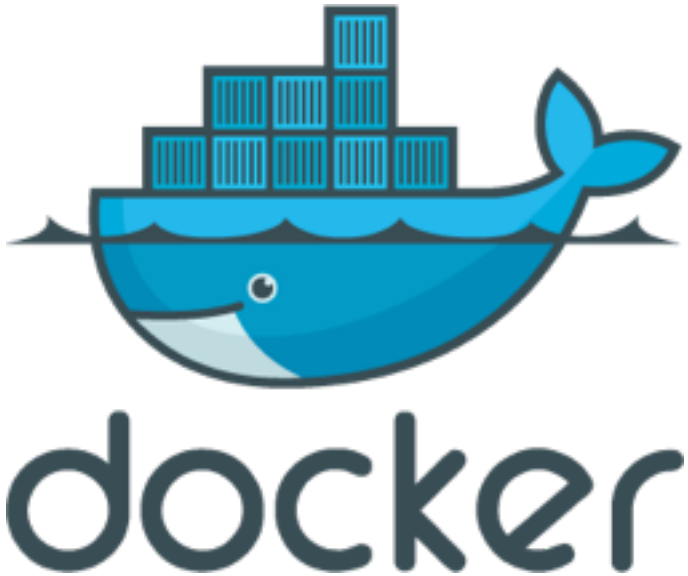
a.k.a. The Javascript Meta-object Protocol

Tom Van Cutsem

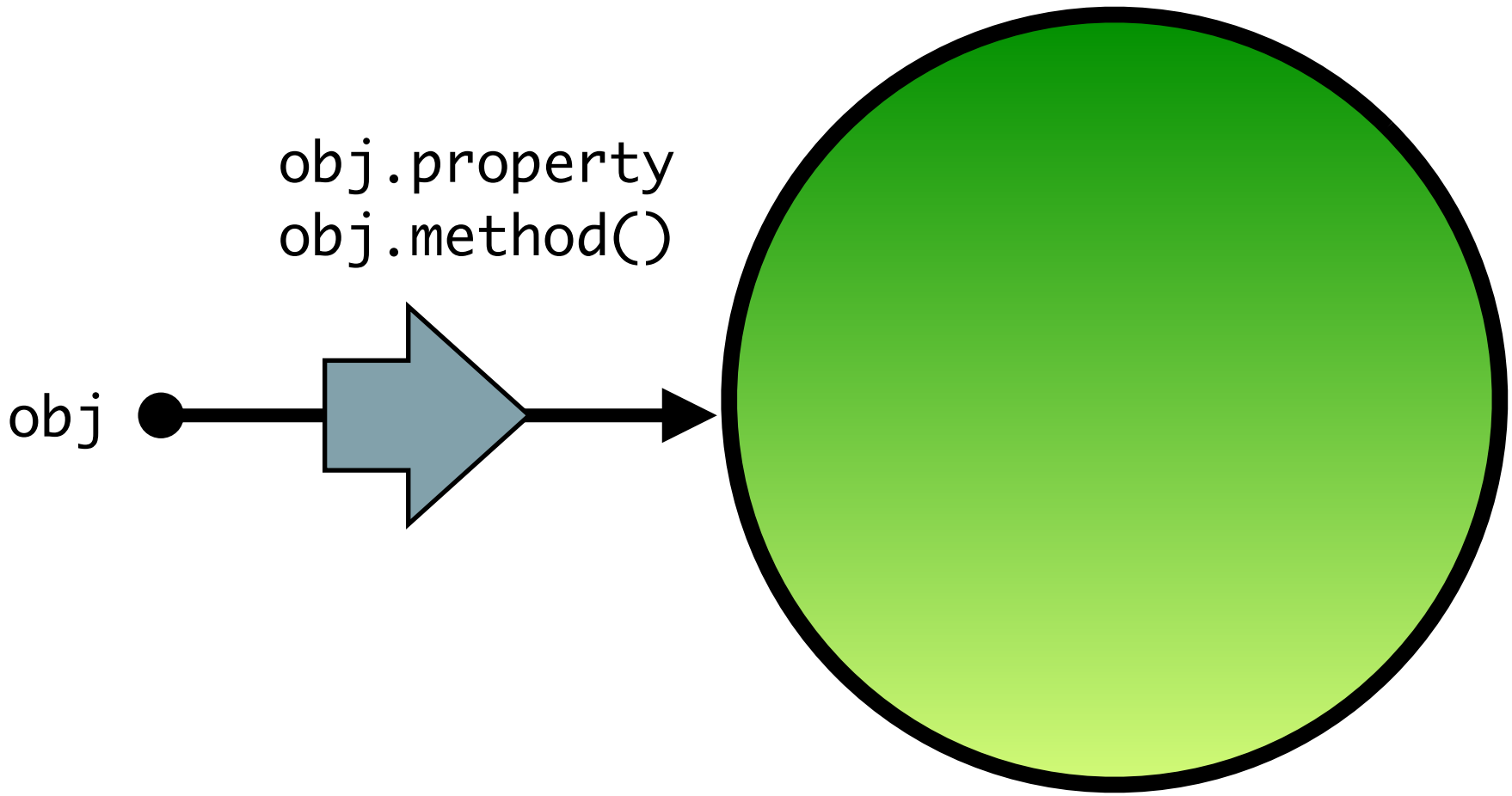
@tvcutsem

A Powerful Idea: Virtualization

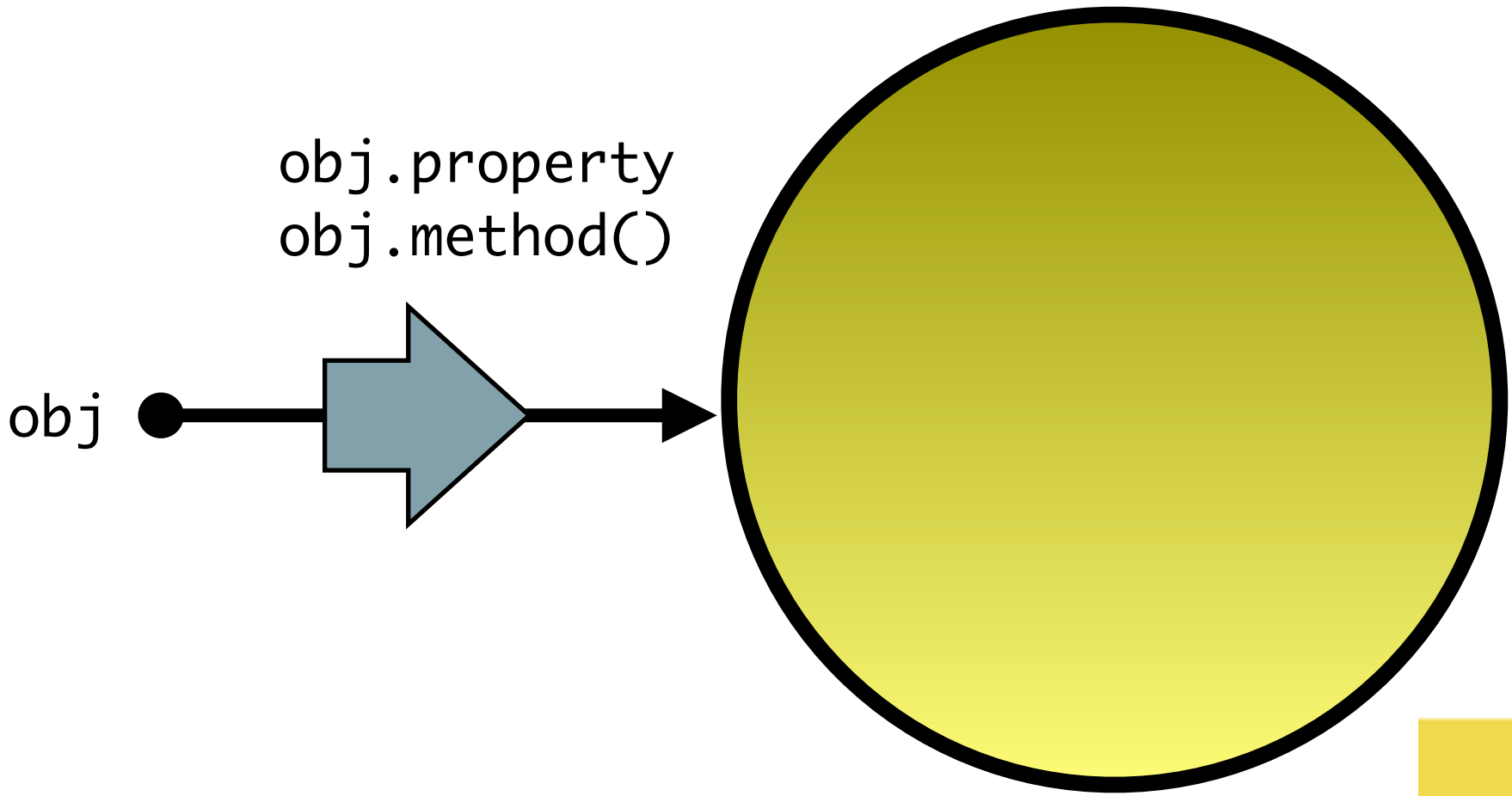
A Powerful Idea: Virtualization



Virtualization in object-oriented languages



Virtualization in JavaScript



Talk Outline

- Brief introduction to Javascript
- Virtualization & Meta-object protocols (MOPs)
- Proxy Objects and how they make Javascript's MOP explicit
- Applications of Proxies

My involvement in JavaScript

- 2004-2008: built up expertise in programming languages research during my PhD
- 2010: Visiting Faculty at Google, joined Caja team
- 2010 - 2014: Joined ECMA TC39 (Javascript standardization committee)

- Actively contributed to the ECMAScript 2015 specification

What developers think about JavaScript

- Lightning talk Gary Bernhardt at CodeMash 2012
- https://www.destroyallsoftware.com/talks/wat

The world's most misunderstood language

Douglas Crockford,
Inventor of JSON

See also: "JavaScript: The World's Most Misunderstood Programming Language" by Doug Crockford at http://www.crockford.com/javascript/javascript.html

The Good Parts

- Functions as first-class objects
- Dynamic objects with prototypal inheritance
- Object literals
- Array literals

The Bad Parts

- Global variables (no modules)
- Var hoisting (no block scope)
- `with` statement
- Implicit type coercion
- …

Good Parts

- Functions (closures, higher-order, first-class)

```
var add = function(a,b) {
  return a+b;
}

add(2,3);
```

```
function makeAdder(a) {
  return function(b) {
    return a+b;
  }
}

makeAdder(2)(3);
```

```
[1,2,3].map(function (x) { return x*x; })
node.addEventListener('click', function (e) { clicked++; })
```

Good Parts

- Objects (no classes, literal syntax, arbitrary nesting)

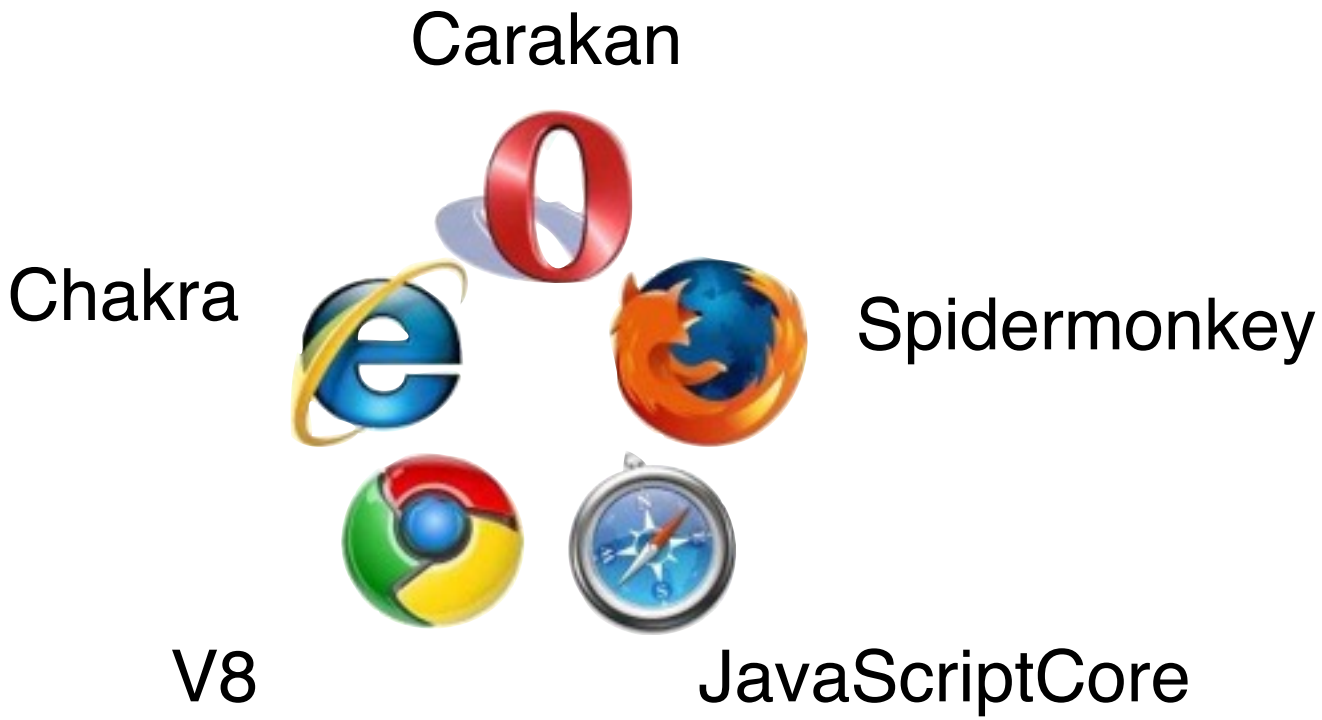
```
var bob = {
  name: “Bob”,
  dob: {
    day: 15,
    month: 03,
    year: 1980
  },
  address: {
    street: “...”,
    number: 5,
    zip: 94040,
    country: “...”
  }
};
```

```
function makePoint(i,j) {
  return {
    x: i,
    y: j,
    toString: function() {
      return ‘(’+ this.x +‘,’+ this.y +‘)’;
    }
  };
}

var p = makePoint(2,3);
var x = p.x;
var s = p.toString();
```

A dynamic language...

```
// computed property access and assignment
obj[“foo”]
obj[“foo”] = 42;

// dynamic method invocation
var f = obj.m;
f.apply(obj, [1,2,3]);

// enumerate an object’s properties
for (var prop in obj) { console.log(prop); }

// dynamically add new properties to an object
obj.bar = baz;

// delete properties from an object
delete obj.foo;
```

ECMAScript: “Standard” JavaScript



ECMAScript: “Standard” JavaScript

~~Carakan~~ V8

Chakra

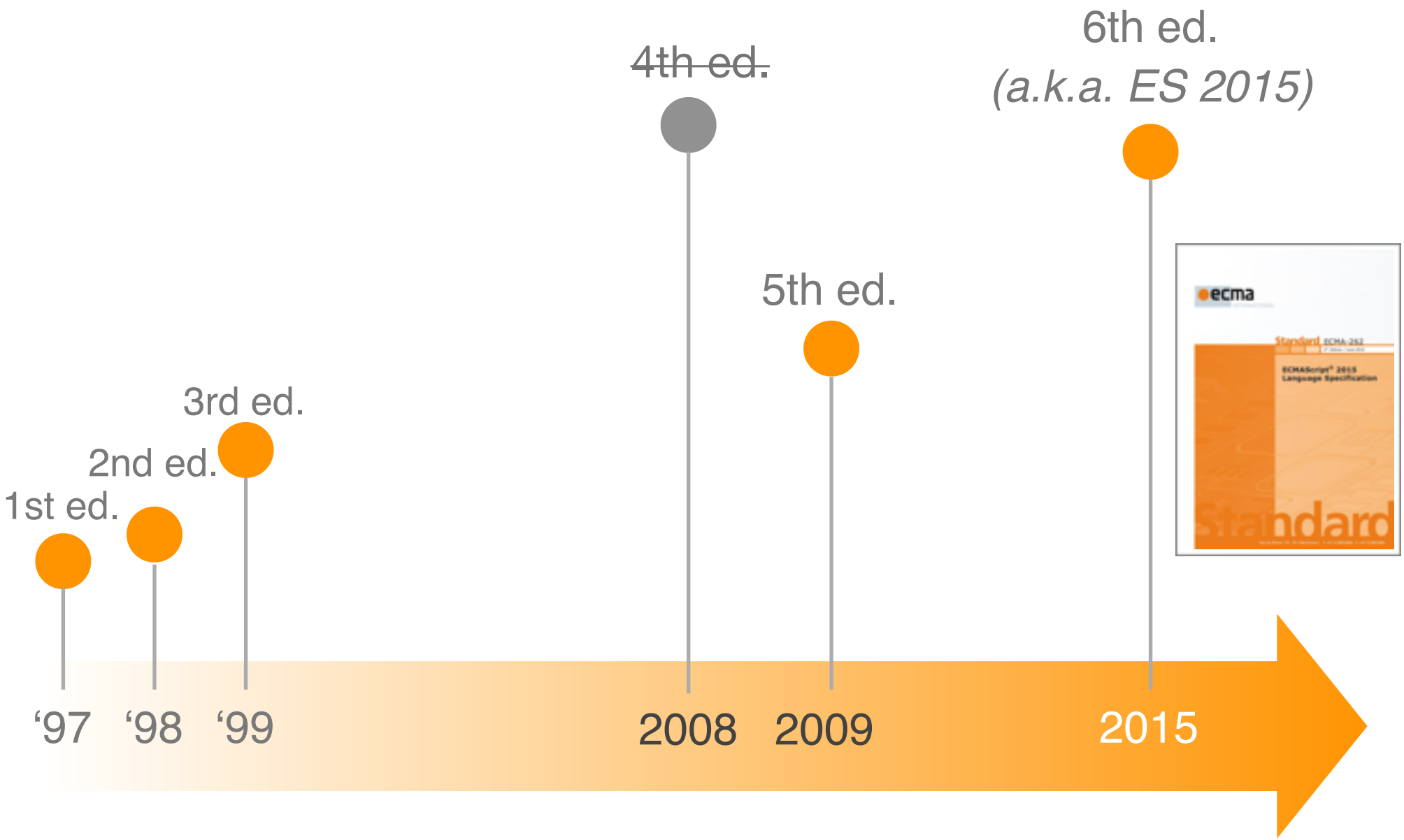
Spidermonkey

V8

JavaScriptCore

node.js™

A brief history of the ECMAScript spec



Functions

- Functions are objects

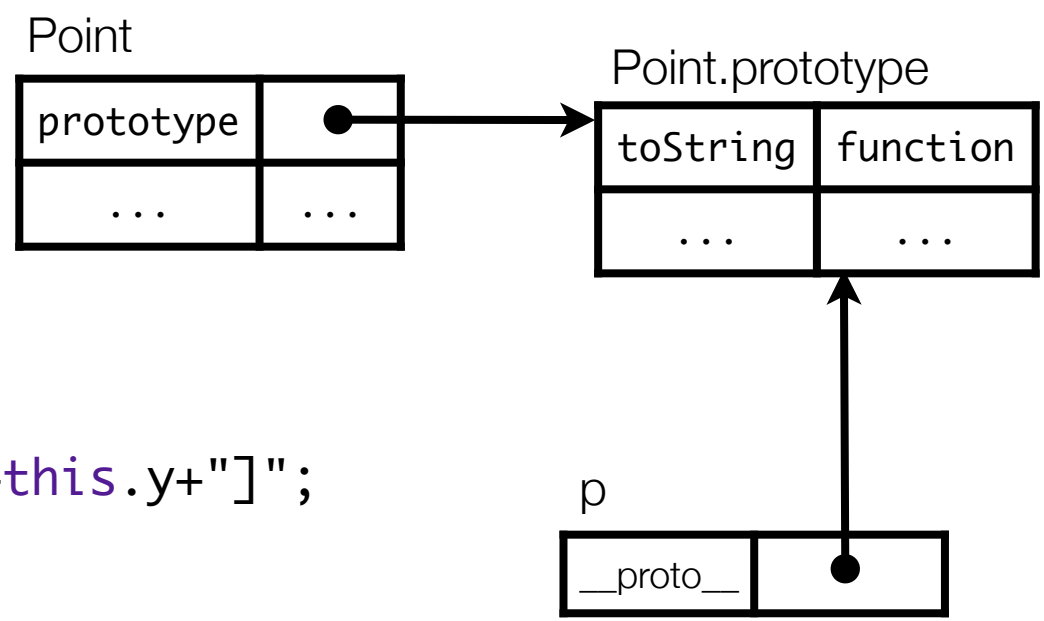
```
function add(x,y) { return x + y; }
add(1,2) // 3

add.apply(undefined, [1,2]) // 3

add.doc = “returns the sum of two numbers”;
```

Objects

- No classes.
- Instead, functions may be used as object constructors.
- All objects have a “prototype” link
  - Lookup of a property on an object traverses the prototype links
  - Similar to inheritance between classes
  - In some implementations, the prototype is an explicit property of the object named `__proto__`

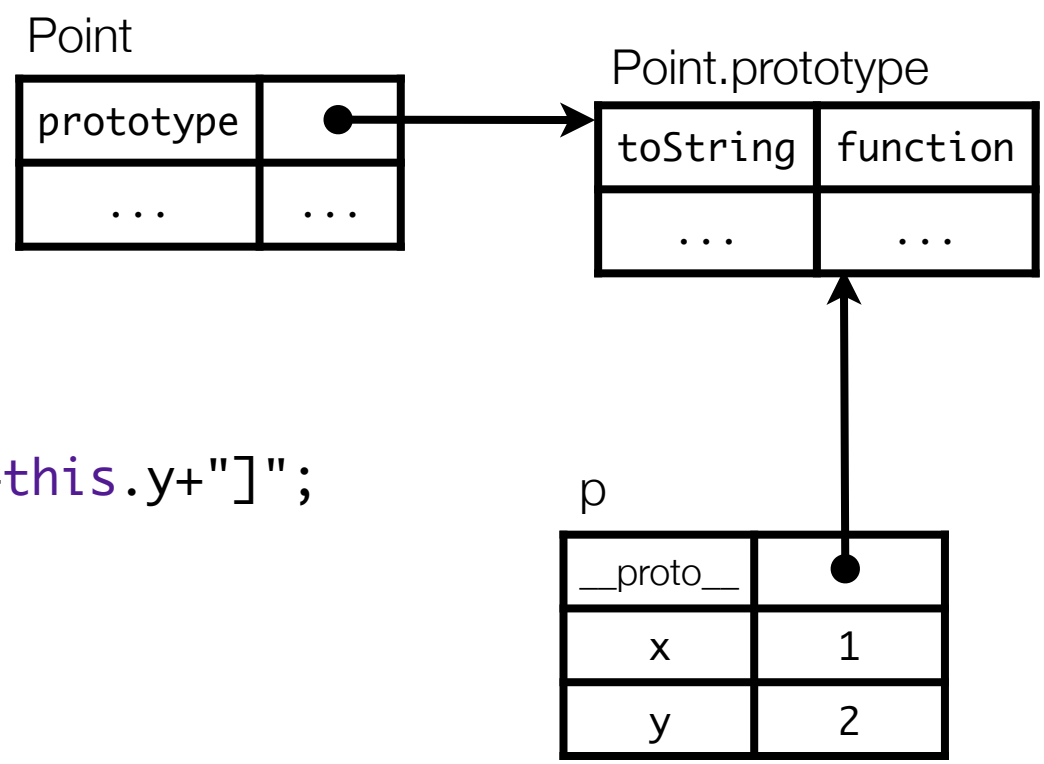
Objects

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}

Point.prototype = {
  toString: function() {
    return "[Point "+this.x+","+this.y+"]";
  }
}

var p = new Point(1,2);
```



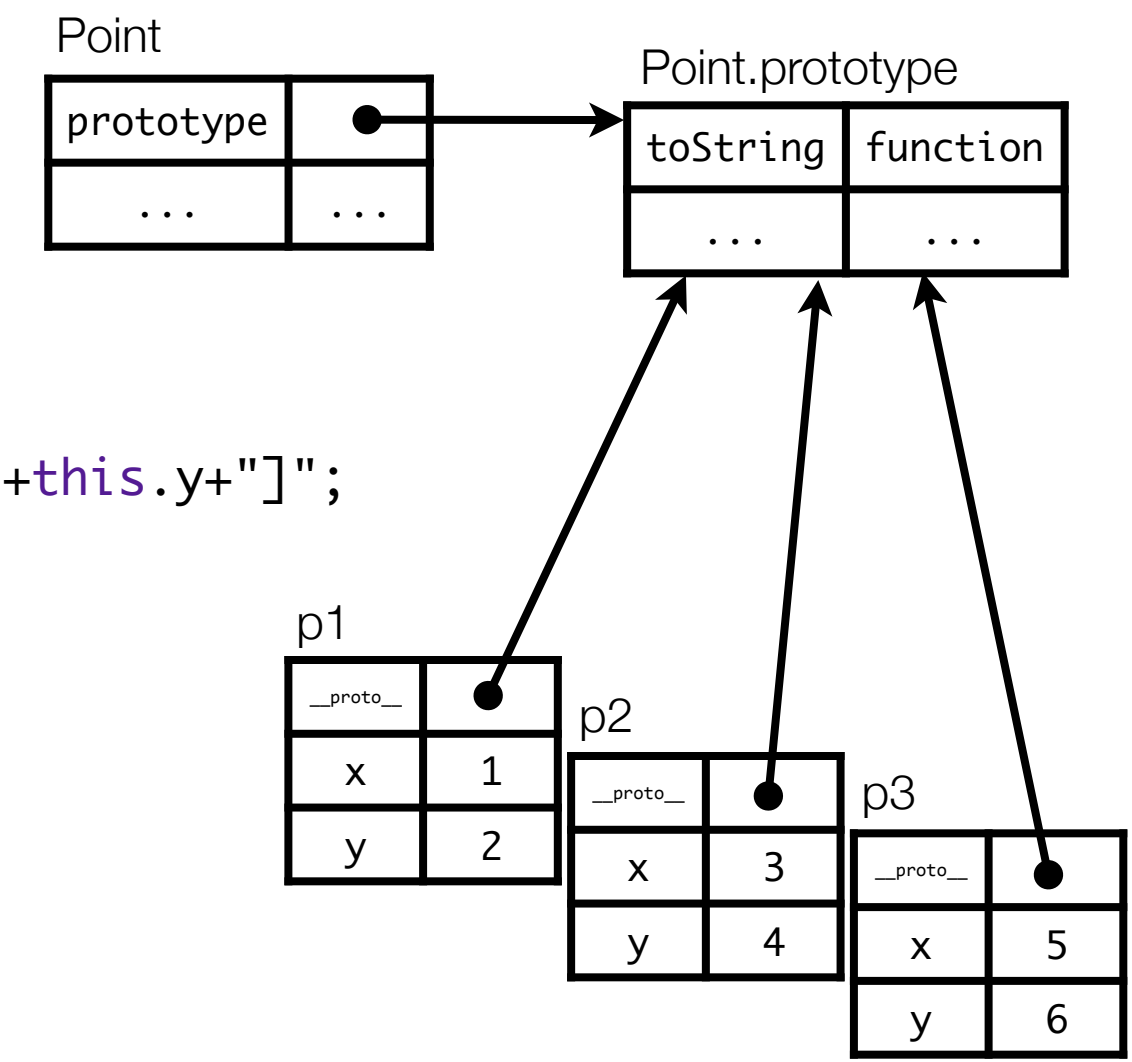
Objects

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}

Point.prototype = {
  toString: function() {
    return "[Point "+this.x+","+this.y+"]";
  }
}

var p = new Point(1,2);
p.x;
p.toString();
```



Objects

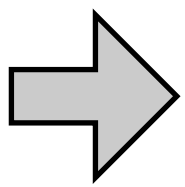
```
function Point(x, y) {
  this.x = x;
  this.y = y;
}

Point.prototype = {
  toString: function() {
    return "[Point "+this.x+","+this.y+"]";
  }
}

var p1 = new Point(1,2);
var p2 = new Point(3,4);
var p3 = new Point(5,6);
```



ECMAScript 6: classes

- ECMAScript 6 classes are really just functions!

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}

Point.prototype = {
  toString: function() {
    return "[Point...]";
  }
}

var p = new Point(1,2);
p.x;
p.toString();
```

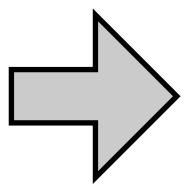


```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }

  toString() {
    return "[Point...]";
  }
}

var p = new Point(1,2);
p.x;
p.toString();
```

ECMAScript 6: classes

- ECMAScript 6 classes are really just functions!

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}

Point.prototype = {
  toString: function() {
    return "[Point...]";
  }
}

var p = new Point(1,2);
p.x;
p.toString();
```



```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }

  toString() {
    return "[Point...]";
  }
}

var p = new Point(1,2);
p.x;
p.toString();
```

Functions / Methods

- Methods of objects are just functions
- When a function is called “as a method”, `this` is bound to the receiver object

```
var obj = {
  offset: 10,
  index: function(x) { return this.offset + x; }
}

obj.index(0); // 10
```

Functions / Methods

- Methods may be “extracted” from objects and used as stand-alone functions

```
var obj = {
  offset: 10,
  index: function(x) { return this.offset + x; }
}

var indexf = obj.index;

otherObj.index = indexf;

indexf(0) // error

indexf.apply(obj, [0]) // 10
```

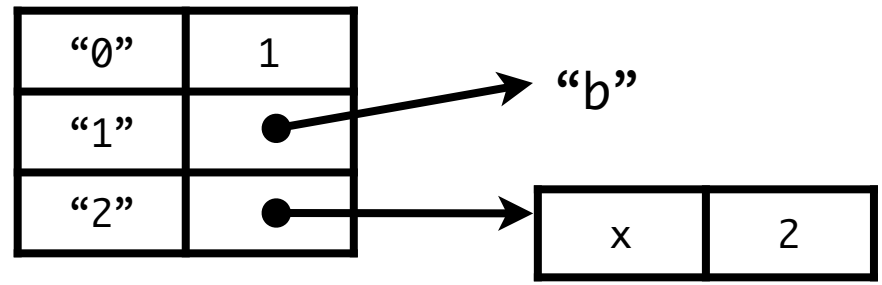
Functions / Methods

- Methods may be “extracted” from objects and used as stand-alone functions

```
var obj = {
  offset: 10,
  index: function(x) { return this.offset + x; }
}

var indexf = obj.index.bind(obj);

indexf(0) // 10
```

Arrays

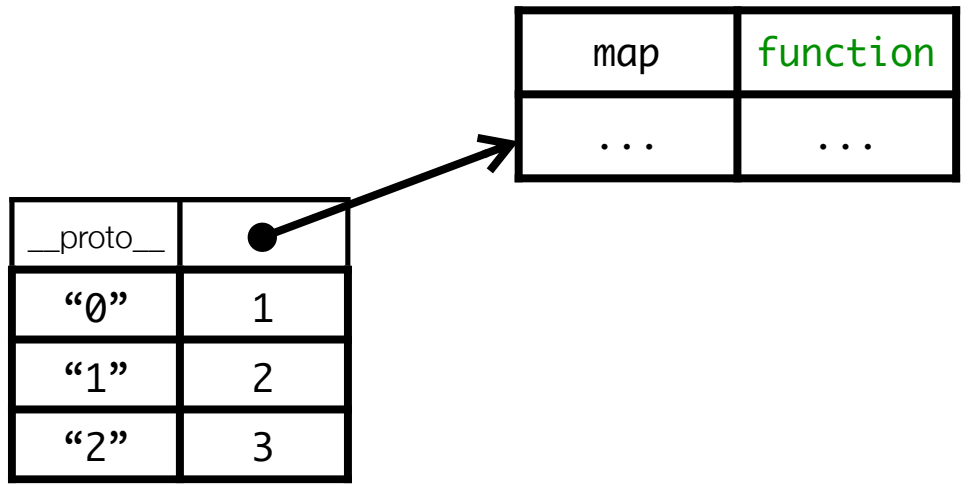
- Arrays are objects, too




```
var a = [1, "b", {x:2}];

a[0]        // 1
a["0"]      // 1
a.length    // 3
a[5] = 0;
a.length    // 6
```

Arrays

- Higher-order functions inherited from `Array.prototype`

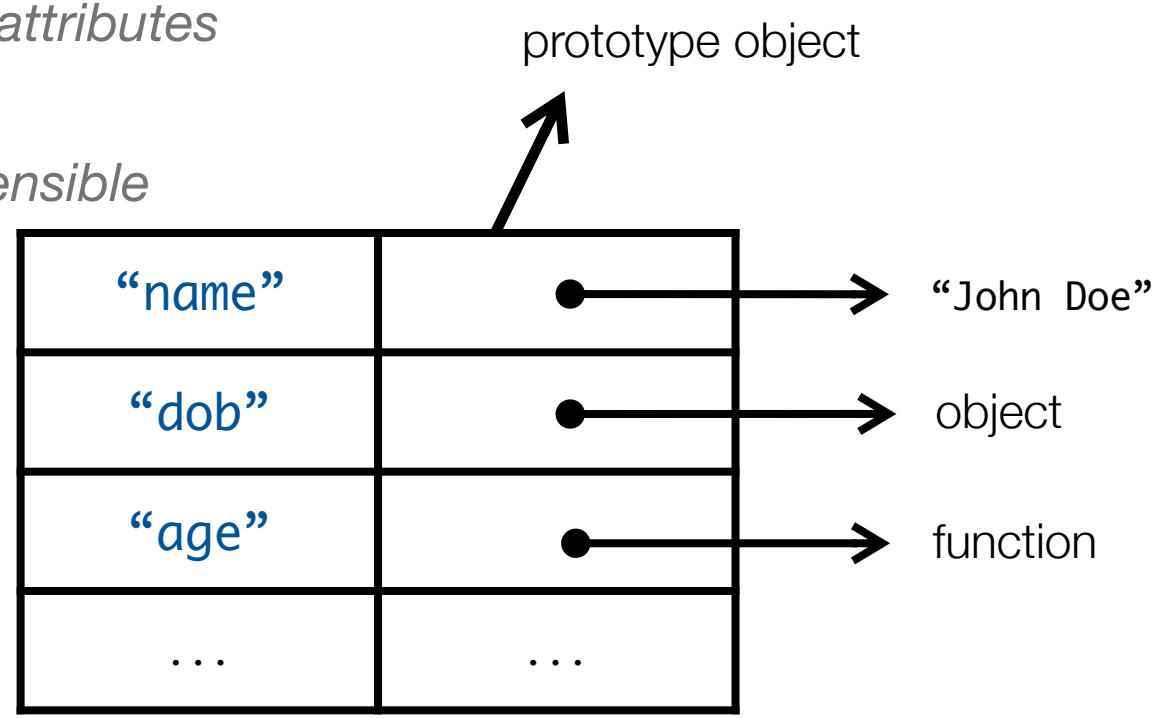



```
[1,2,3].map(function (x) { return x + 1; });

[1,2,3].reduce(function(sum, x) { return sum + x; }, 0);

...
```

Javascript's Object Model

- A Javascript object is a map of strings to values + a prototype pointer
- Just a first approximation:
  - properties have hidden *attributes*
  - objects can be *non-extensible*

```
{ name: "John Doe",
   dob: {...},
   age: function(){...},
   ...  };
```



Property attributes

```
var point =
  { x: 0,
    y: 0  };
```

```
Object.getOwnPropertyDescriptor(point, 'x');
  { value: 0,
    writable: true,
    enumerable: true,
    configurable: true }
```

```
Object.defineProperty(point, 'x',
  { value: 1,
    writable: false,
    enumerable: false,
    configurable: false });
```

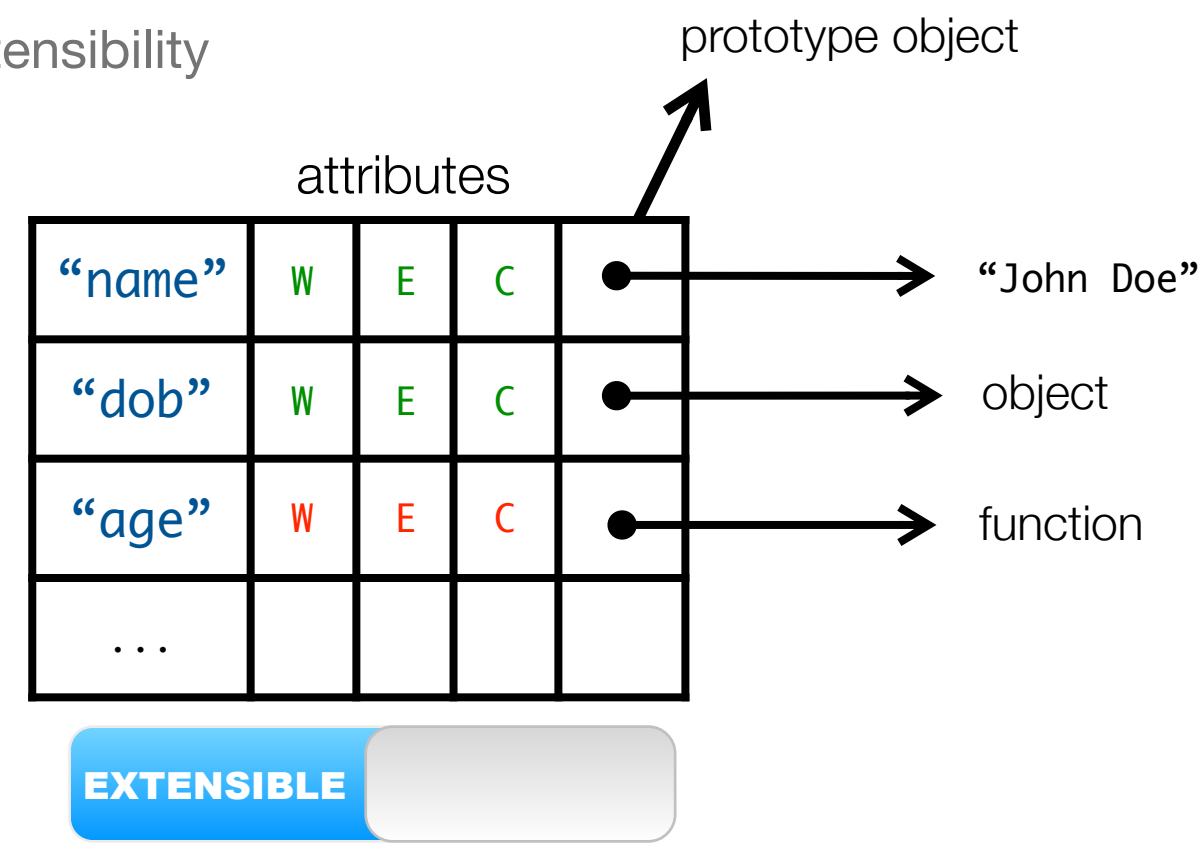
Tamper-proof Objects

```
var point =
  { x: 0,
    y: 0  };


  Object.preventExtensions(point);
  point.z = 0; // error: can't add new properties


  Object.seal(point);
  delete point.x; // error: can't delete properties


  Object.freeze(point);
  point.x = 7; // error: can't assign properties
```

Javascript's Object Model Revisited

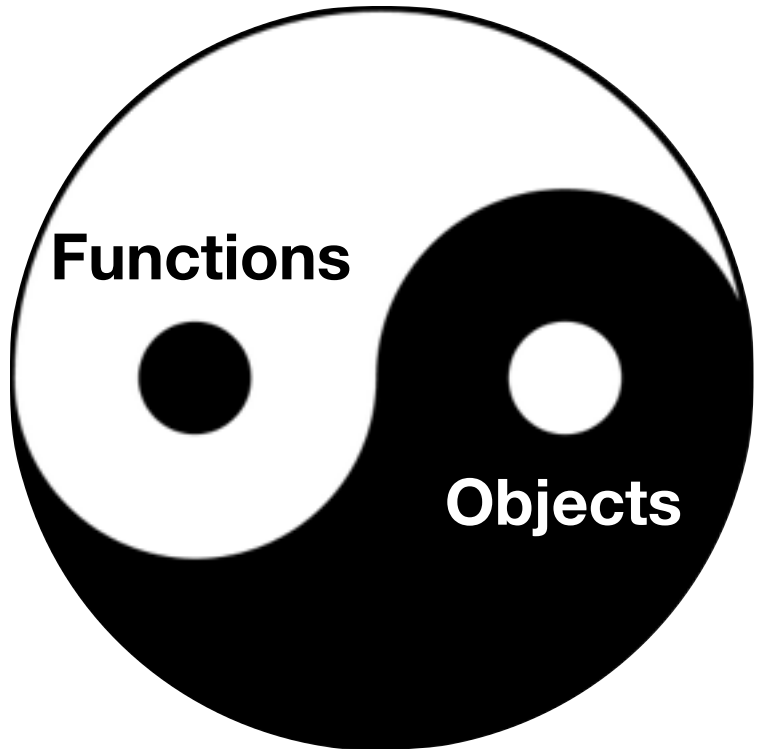
- A Javascript object is a map of strings to *property descriptors*
  - + a prototype pointer
  - + a flag indicating extensibility

```
{ name: "John Doe",
    dob: {...},
    age: function(){...},
    ...  };
```



Summary so far

- Javascript: “a Lisp in C’s clothing” (D. Crockford)
- Good parts: functions, object literals, flexible prototypes
- Beware of and avoid the “bad parts”
- Functions and objects work well together
- Javascript object =
  property map
  + prototype link
  + extensible flag

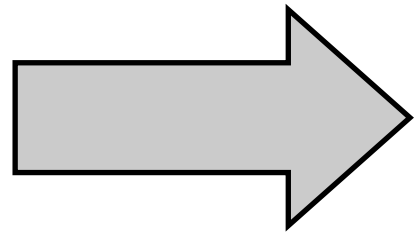


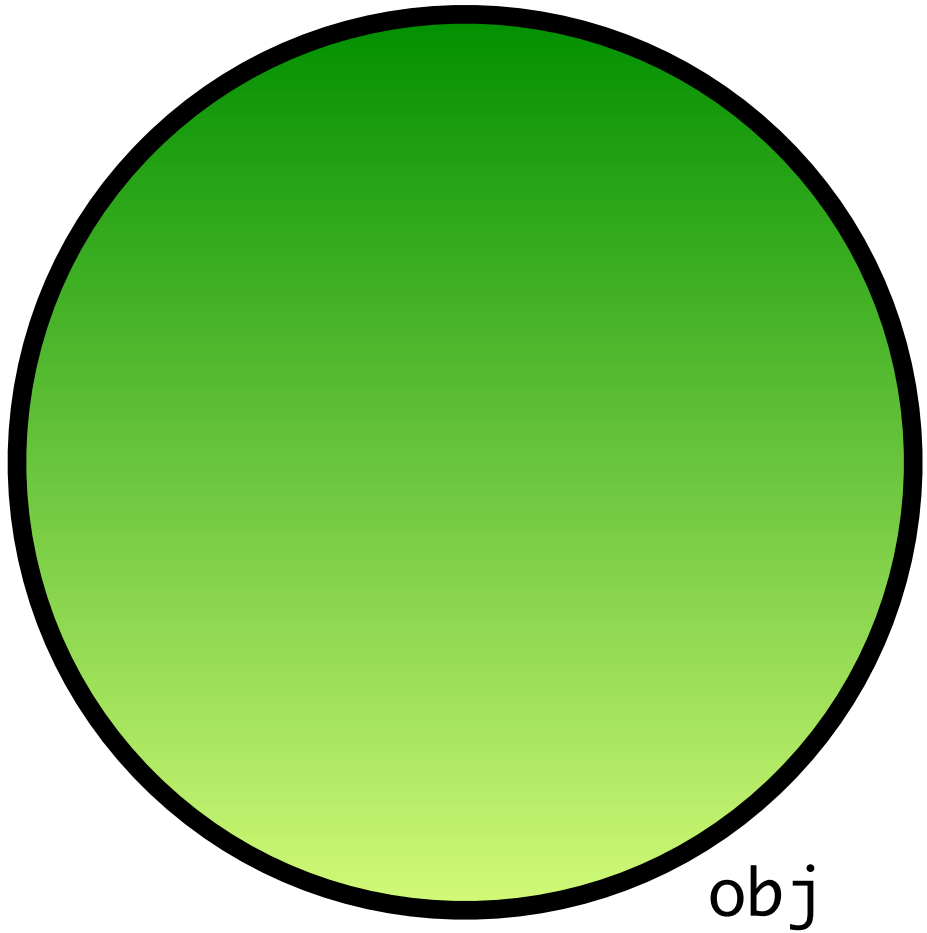
Virtualization & Meta-Object Protocols

Introspection vs. Intercession

querying an object

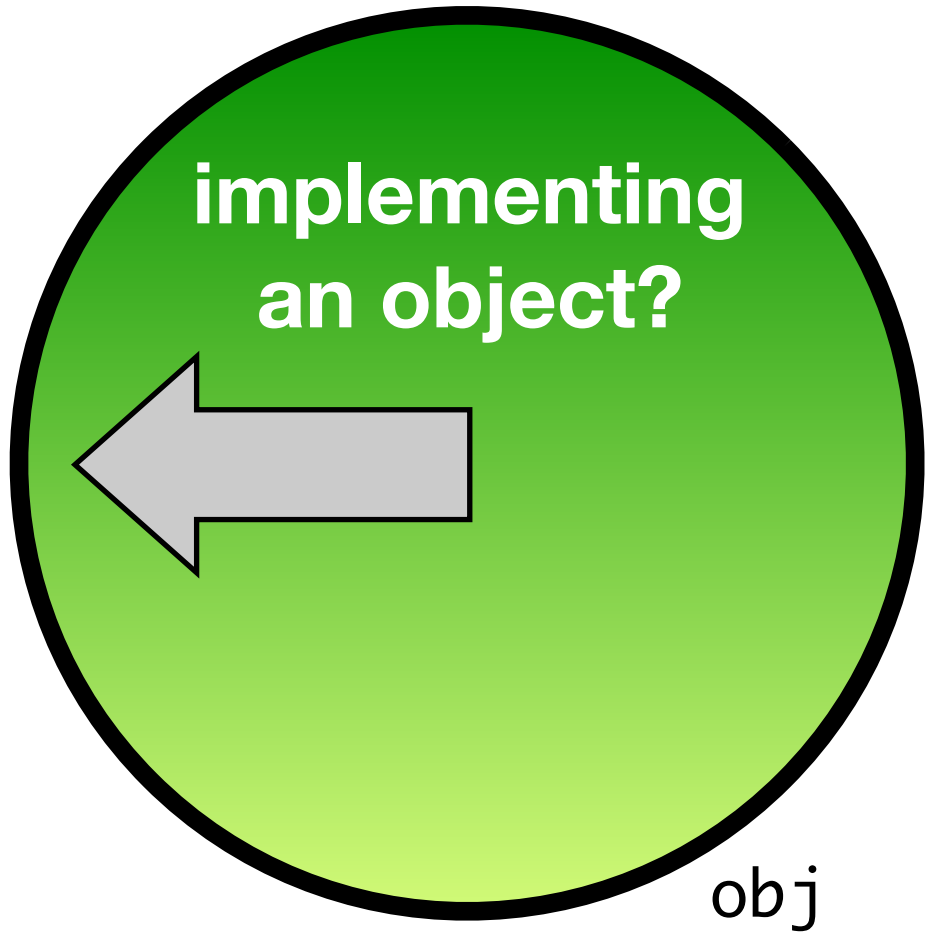
acting upon an object






```
obj[“x”]
delete obj.x;
Object.getOwnPropertyDescriptor(obj,‘x’);
```

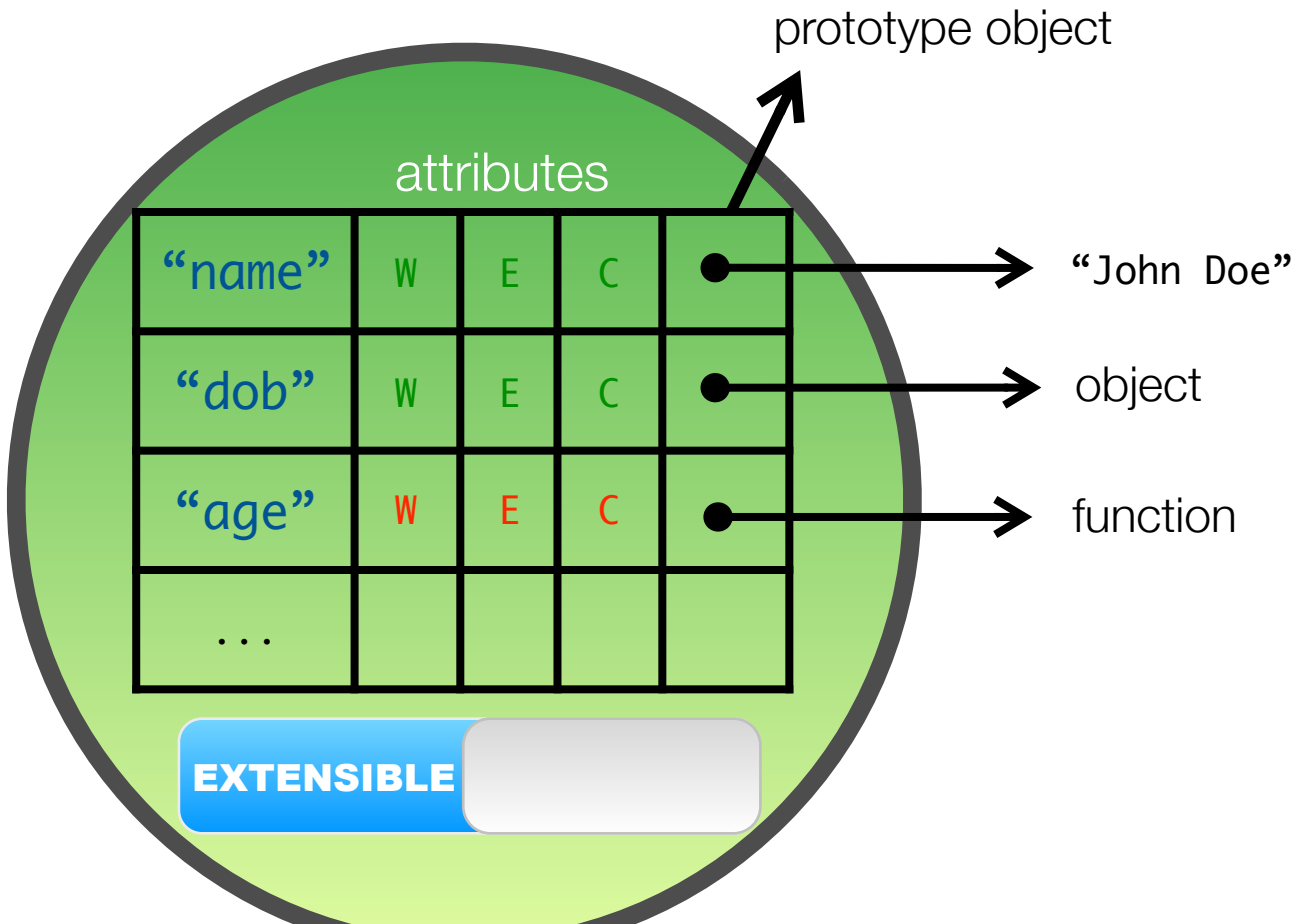
Introspection vs. Intercession




```
obj[“x”]
delete obj.x;
Object.getOwnPropertyDescriptor(obj,‘x’);
```

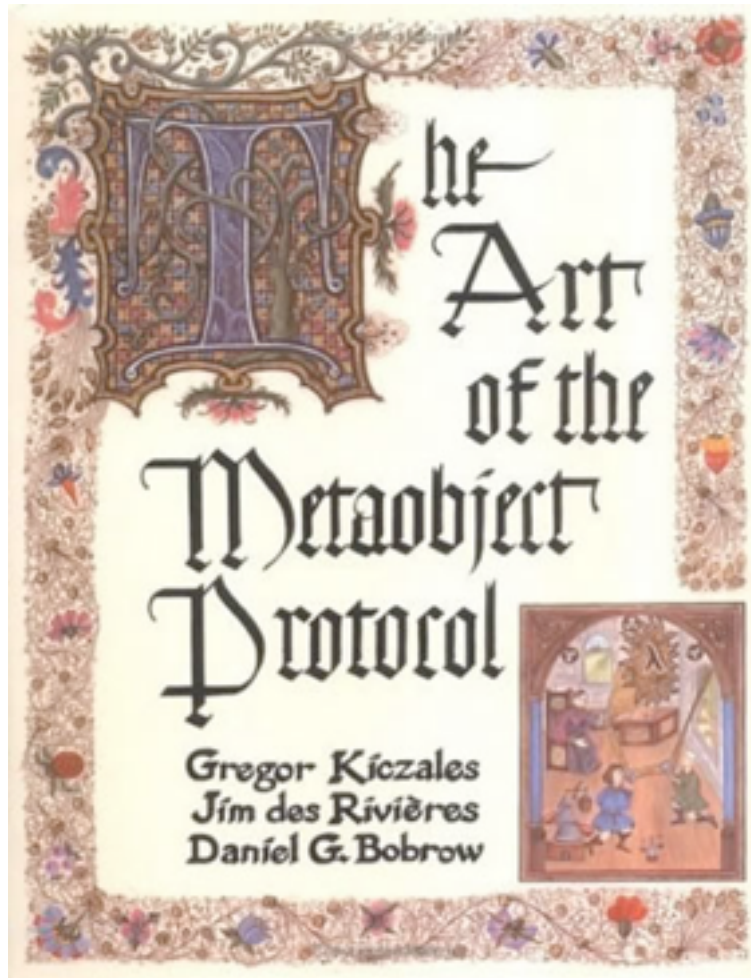
Javascript's Object Model: Recap

- The object model defines the "interface" of an object
- Any "sensible" implementation of this model is a valid Javascript object



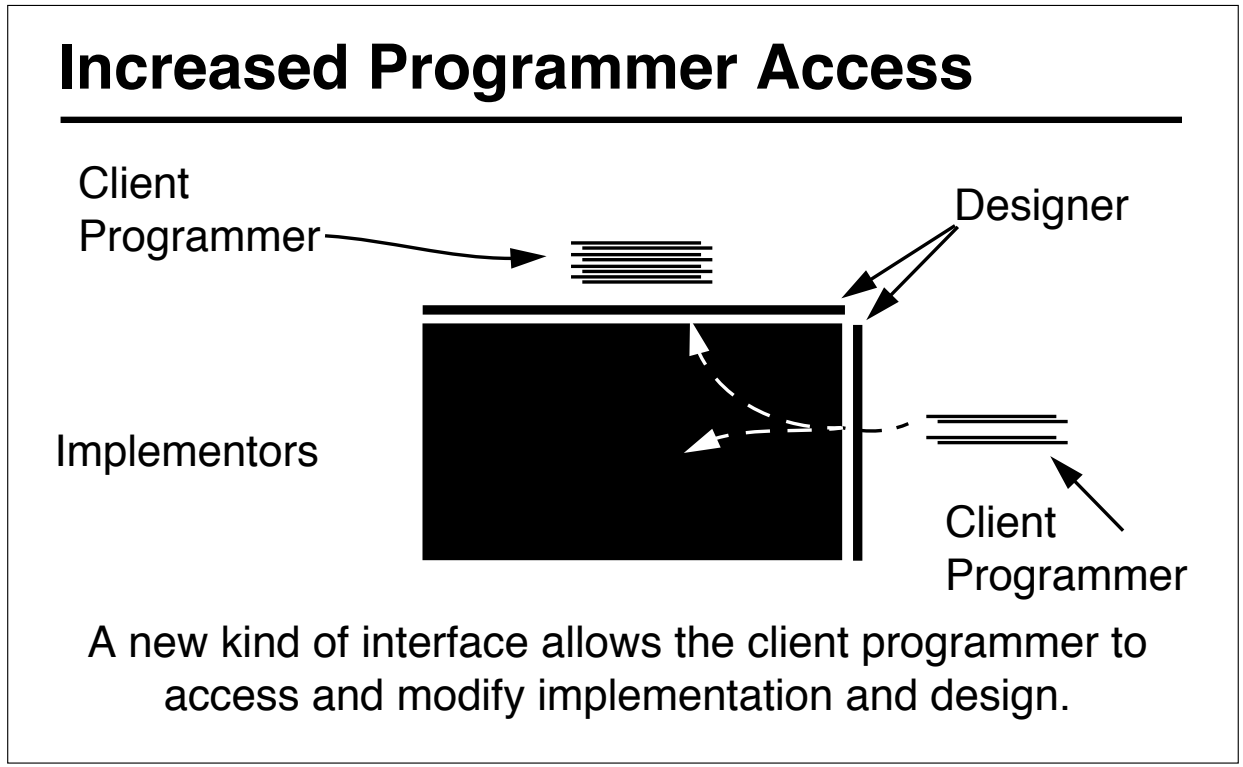
Meta-object protocols

- Kiczales, early ‘90s
- Using OOP to structure the meta-level
- Common Lisp Object System (CLOS)
- Precursor to AOP



“Open implementations” philosophy

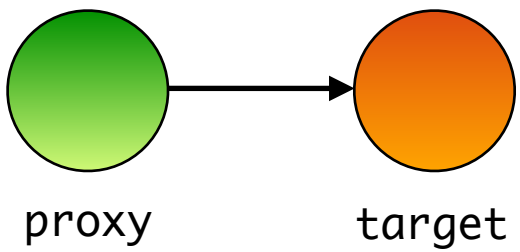
- Kiczales & Paepcke, early ‘90s



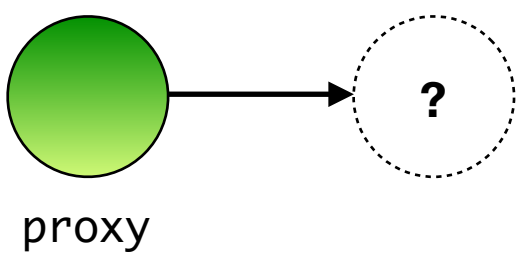
(Kiczales & Paepcke, Open Implementations & Metaobject Protocols)

Why implement your own object?

- Generic wrappers around existing objects: access control wrappers, tracing, profiling, contracts, taint tracking, ...



- Virtual objects: remote objects, mock objects, persistent objects, futures/promises, lazily initialized objects, ...



ECMAScript 5 does not support this

- ECMAScript 5 reflection API:
  - powerful control over structure of objects
  - limited control over behavior of objects
- Can’t intercept method calls, property access, ...
- Can’t implement ‘virtual’ properties

Limited intercession in some implementations

- non-standard `__noSuchMethod__` hook in Firefox
- modelled after Smalltalk's `doesNotUnderstand:` method

```
function makeProxy(target) {
  return {
    __noSuchMethod__: function(name, args) {
      return target[name].apply(target, args);
    }
  };
}
```

__noSuchMethod__

- not “stratified” (part of base-level object interface)
- limited intercession (intercepts only missing method calls)

```
var p = makeProxy({foo: 42});
‘foo’ in p          // false
p.__noSuchMethod__ // reveals the method
for (var name in p) {
  // reveals ‘__noSuchMethod__’ but not ‘foo’
}
```

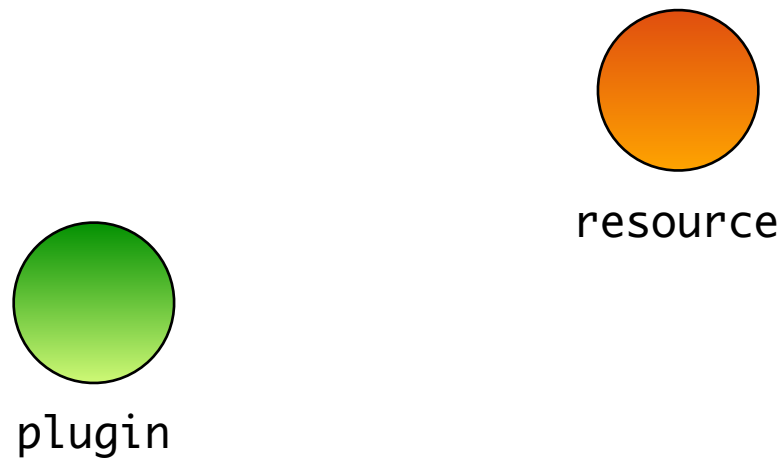
Proxies: virtual objects

Proxies

- Objects that “look and feel” like normal objects, but whose behavior is controlled by *another* Javascript object
- Part of a new reflection API for ECMAScript 6
- Think `java.lang.reflect.Proxy` on steroids

Proxy example: revocable references

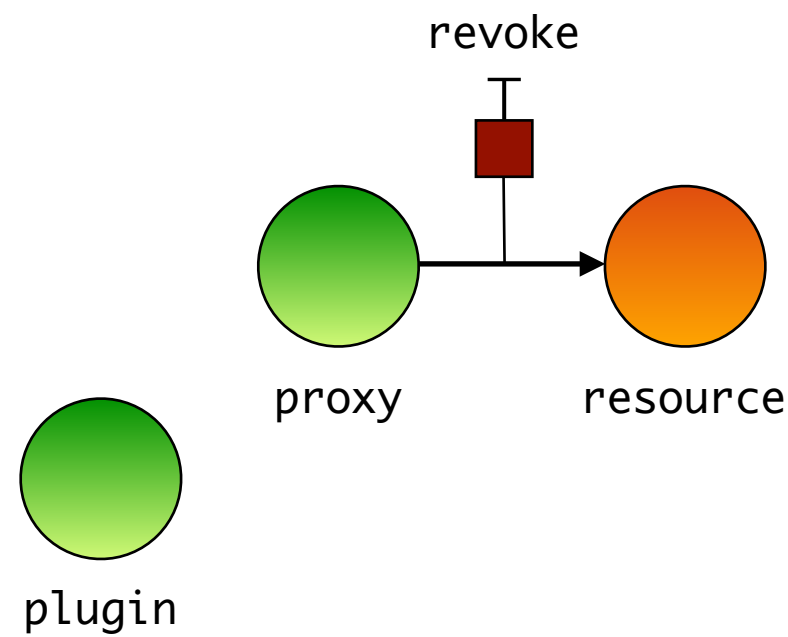
- Provide temporary access to a resource
- Useful for explicit memory management or expressing security policy



Proxy example: revocable references

- Provide temporary access to a resource
- Useful for explicit memory management or expressing security policy

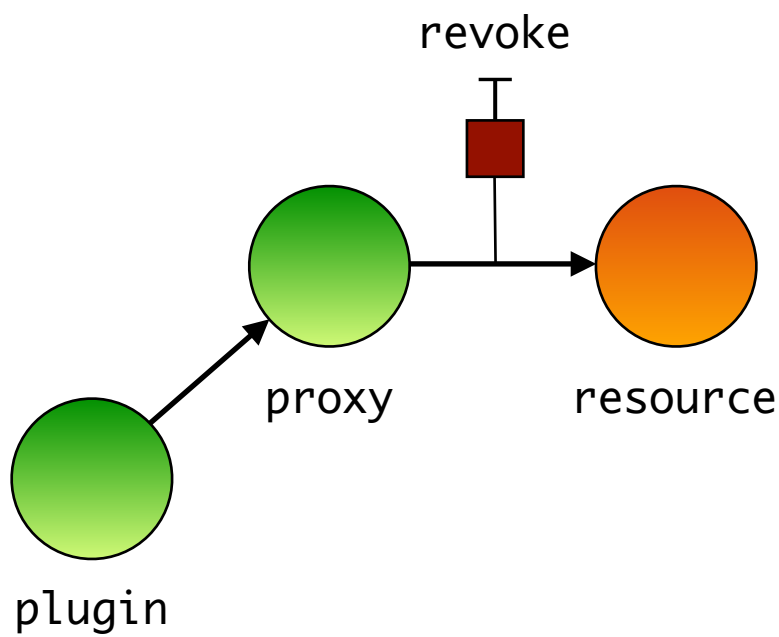
```
var {proxy,revoke} = makeRevocable(resource);
```



Proxy example: revocable references

- Provide temporary access to a resource
- Useful for explicit memory management or expressing security policy

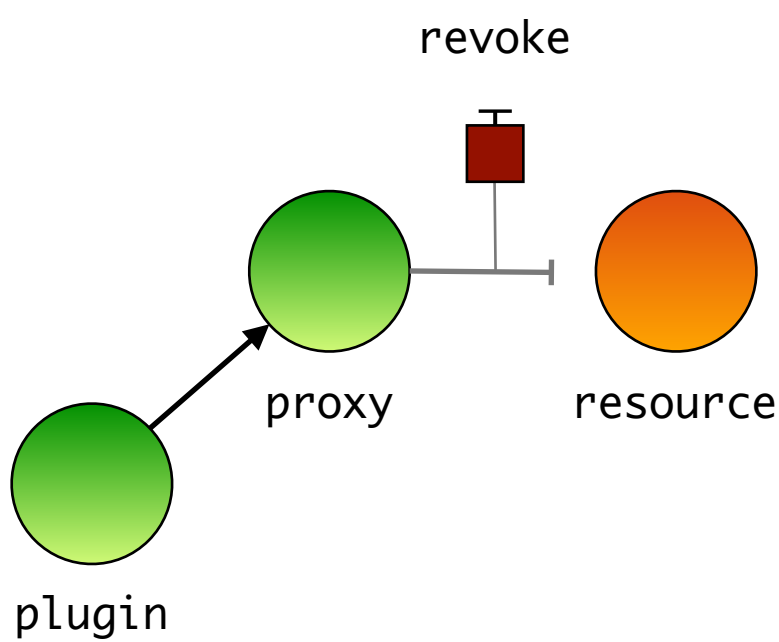
```
var {proxy,revoke} = makeRevocable(resource);

plugin.give(proxy)
```



Proxy example: revocable references

- Provide temporary access to a resource
- Useful for explicit memory management or expressing security policy

```
var {proxy,revoke} = makeRevocable(resource);

plugin.give(proxy)

...
revoke();
```



Revocable references

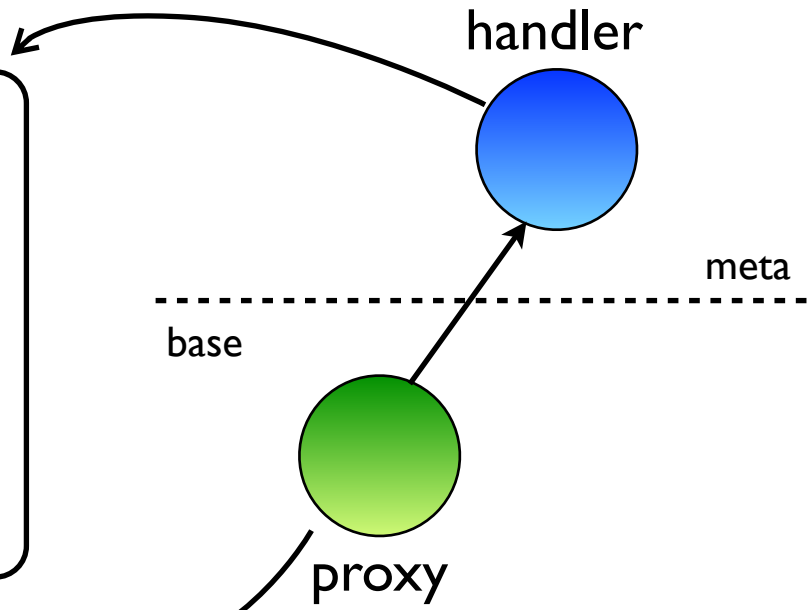
```
function makeRevocable(target) {
  var enabled = true;
  var proxy = new Proxy(target, {

  });
  return {
     proxy: proxy,
    revoke: function() { enabled = false; }
  }
}
```

Revocable references

```
function makeRevocable(target) {
  var enabled = true;
  var proxy = new Proxy(target, {
    get: function(tgt, name) {
      if (!enabled) throw Error("revoked")
      return target[name];
    },
    set: function(tgt, name, val) {
      if (!enabled) throw Error("revoked")
      target[name] = val;
    },
    ...
  });
  return {
     proxy: proxy,
    revoke: function() { enabled = false; }
  }
}
```

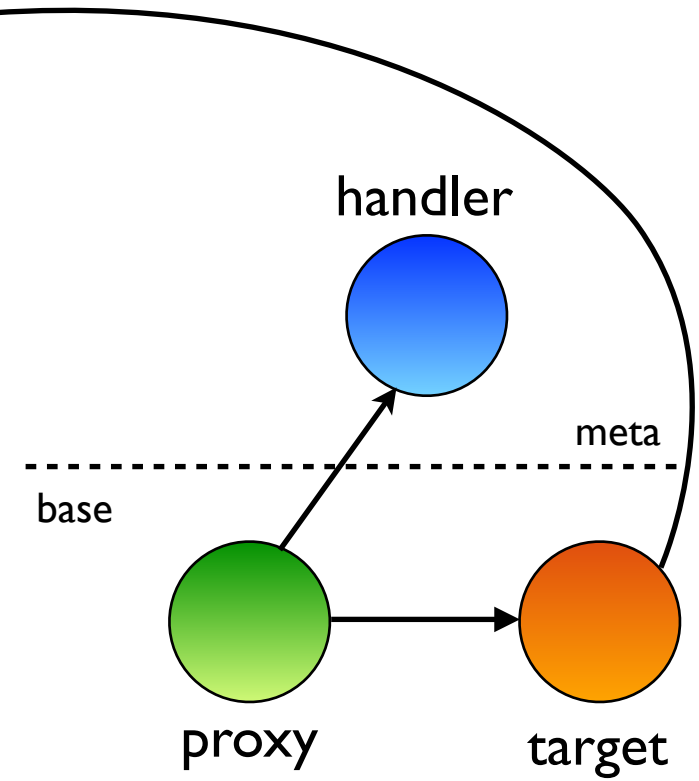
Revocable references

```
function makeRevocable(target) {
  var enabled = true;
  var proxy = new Proxy(target, {
    get: function(tgt, name) {
      if (!enabled) throw Error("revoked")
      return target[name];
    },
    set: function(tgt, name, val) {
      if (!enabled) throw Error("revoked")
      target[name] = val;
    },
    ...
  });
  return {
     proxy: proxy,
    revoke: function() { enabled = false; }
  }
}
```

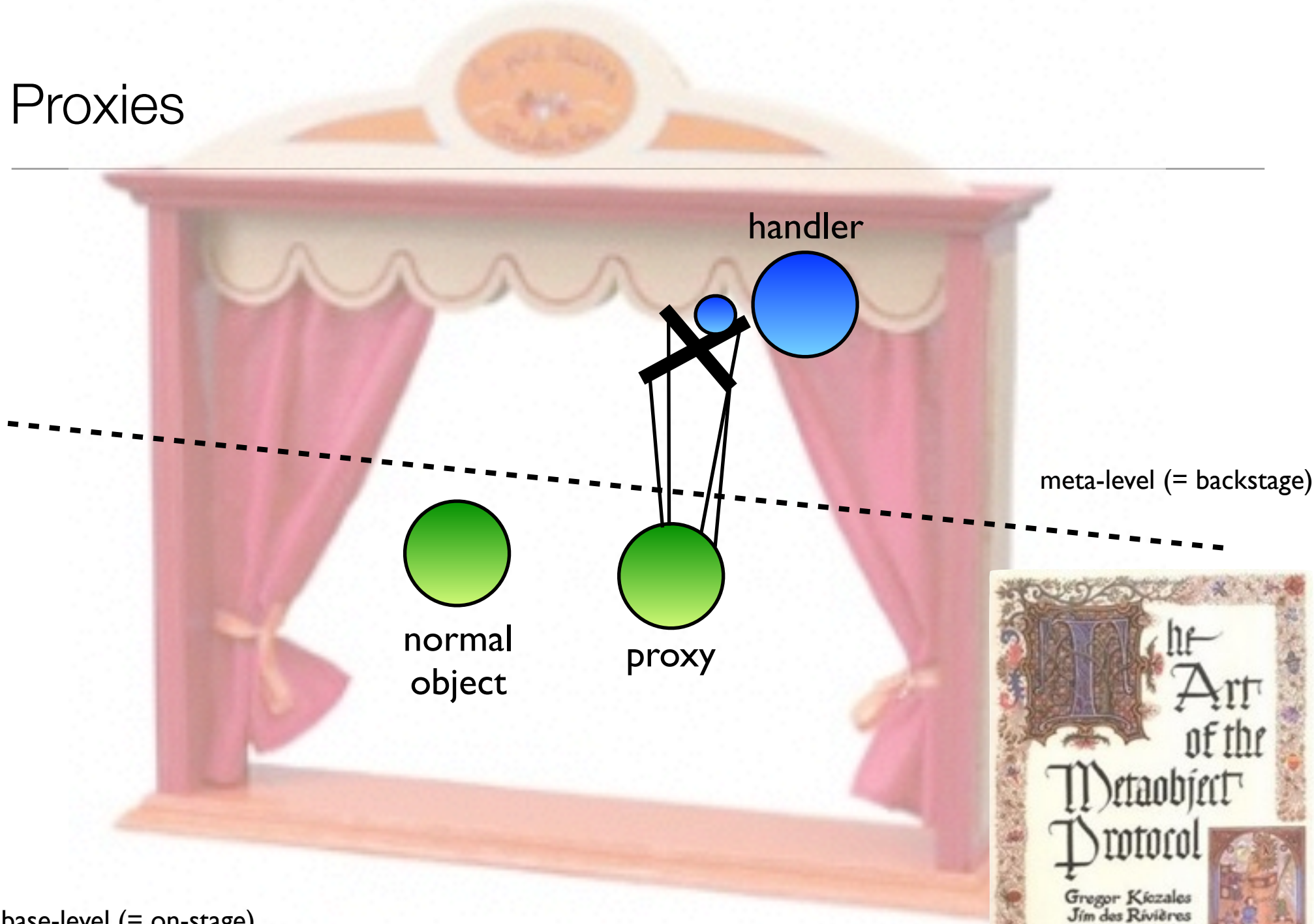


Revocable references

```
function makeRevocable(target) {
  var enabled = true;
  var proxy = new Proxy(target, {
    get: function(tgt, name) {
      if (!enabled) throw Error("revoked")
      return target[name];
    },
    set: function(tgt, name, val) {
      if (!enabled) throw Error("revoked")
      target[name] = val;
    },
    ...
  });
  return {
     proxy: proxy,
    revoke: function() { enabled = false; }
  }
}
```



Proxies


handler
meta-level (= backstage)
normal object
proxy
base-level (= on-stage)

The Art of the Metaobject Protocol
Gregor Kiczales
Jim des Rivières
Daniel G. Bobrow

Stratified API

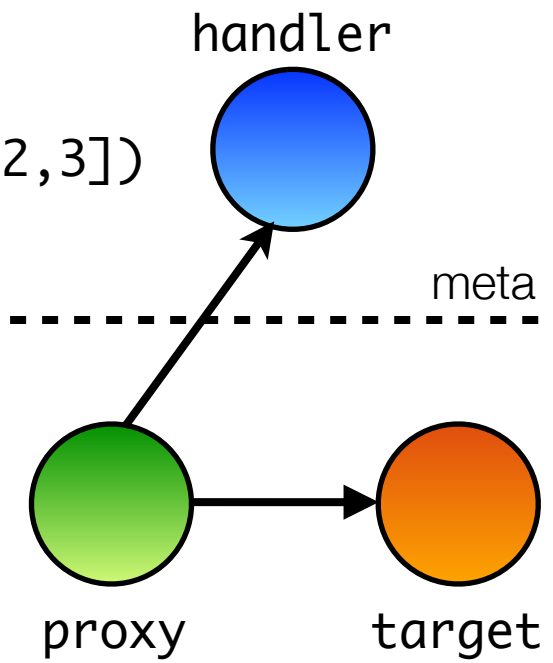
```
var proxy = new Proxy(target, handler);
```

```
handler.get(target, ‘foo’)
handler.set(target, ‘foo’, 42)
handler.get(target, ‘foo’).apply(proxy,[1,2,3])
handler.get(target, ‘get’)
```

meta

base

```
proxy.foo
proxy.foo = 42
proxy.foo(1,2,3)
proxy.get
```

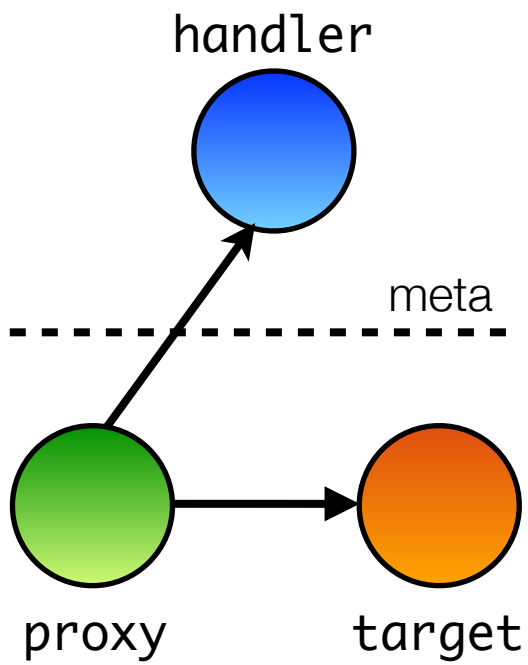


Not just property access

```
var proxy = new Proxy(target, handler);

handler.has(target,‘foo’)

handler.deleteProperty(target,‘foo’)

var props = handler.enumerate(target);
for (var p in props) { ... }

handler.defineProperty(target, ‘foo’, pd)
```

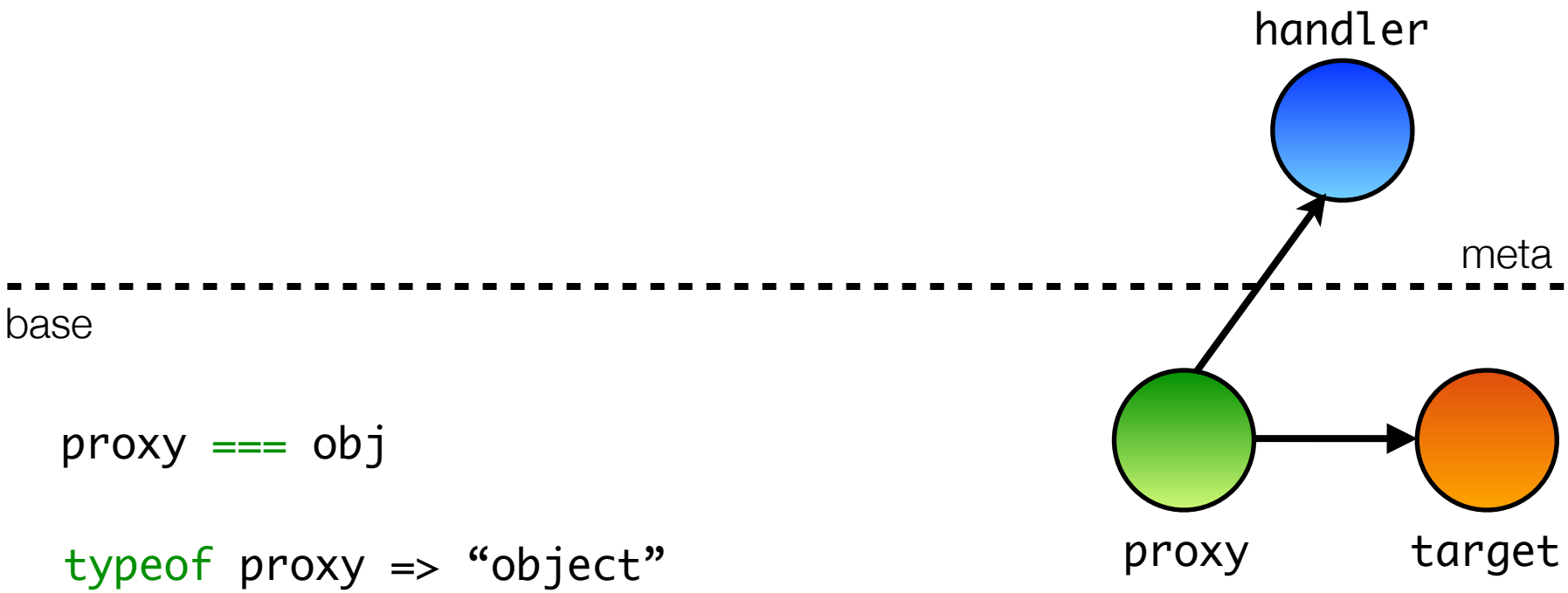
base

```
‘foo’ in proxy

delete proxy.foo

for (var p in proxy) { ... }

Object.defineProperty(proxy,‘foo’, pd)
```



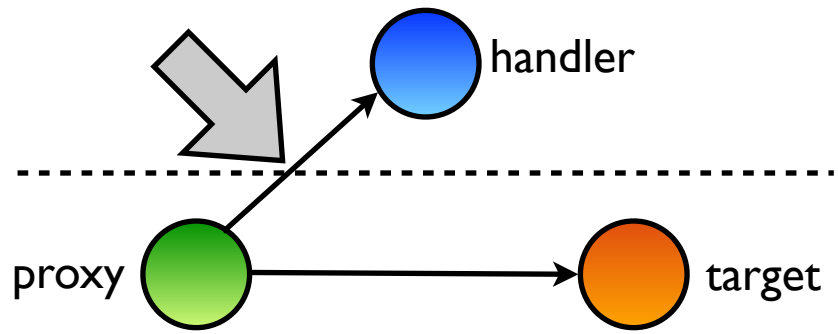
But not quite everything either

```
var proxy = new Proxy(target, handler);
```




```
proxy === obj

typeof proxy => “object”
```

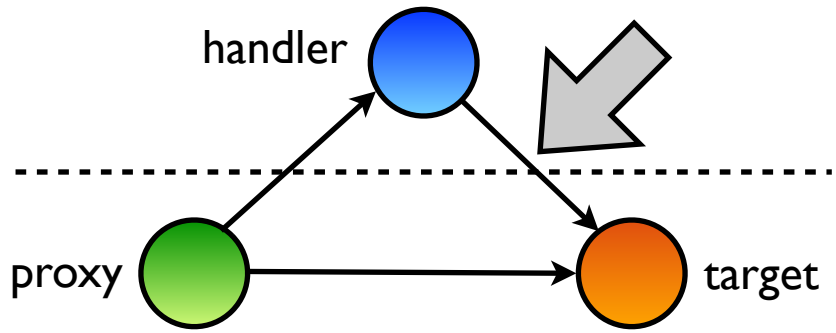
Full handler API (16 traps)




```
Object.getOwnPropertyDescriptor(proxy,name)
Object.defineProperty(proxy,name,pd)
Object.getOwnPropertyNames(proxy)
delete proxy.name
for (name in proxy) { ... }
for (name in Object.create(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(target,name)
handler.defineProperty(target,name,pd)
handler.getOwnPropertyNames(target)
handler.deleteProperty(target,name)
handler.iterate(target)
handler.enumerate(target)
handler.{freeze|seal|preventExtensions}(target)
handler.has(target,name)
handler.hasOwn(target,name)
handler.keys(target)
handler.get(target,name,receiver)
handler.set(target,name,value,receiver)
handler.apply(target,receiver,args)
handler.construct(target,args)
```

Reflect API




```
handler.getOwnPropertyDescriptor(target,name)
handler.defineProperty(target,name,pd)
handler.getOwnPropertyNames(target)
handler.deleteProperty(target,name)
handler.iterate(target)
handler.enumerate(target)
handler.{freeze|seal|preventExtensions}(target)
handler.has(target,name)
handler.hasOwn(target,name)
handler.keys(target)
handler.get(target,name,receiver)
handler.set(target,name,value,receiver)
handler.apply(target,receiver,args)
handler.construct(target,args)
```

```
Reflect.getOwnPropertyDescriptor(target,name)
Reflect.defineProperty(target,name,pd)
Reflect.getOwnPropertyNames(target)
Reflect.deleteProperty(target,name)
Reflect.iterate(target)
Reflect.enumerate(target)
Reflect.{freeze|seal|preventExtensions}(target)
Reflect.has(target,name)
Reflect.hasOwn(target,name)
Reflect.keys(target)
Reflect.get(target,name,receiver)
Reflect.set(target,name,value,receiver)
Reflect.apply(target,receiver,args)
Reflect.construct(target,args)
```

Another example: profiling

```
function makeProfiler(target) {
  var count = new Map();
  return {
    proxy: new Proxy(target, {
      get: function(target, name, receiver) {
        count.set(name, (count.get(name) || 0) + 1);
        return Reflect.get(target, name, receiver);
      }
    }),
    stats: count;
  }
}
```

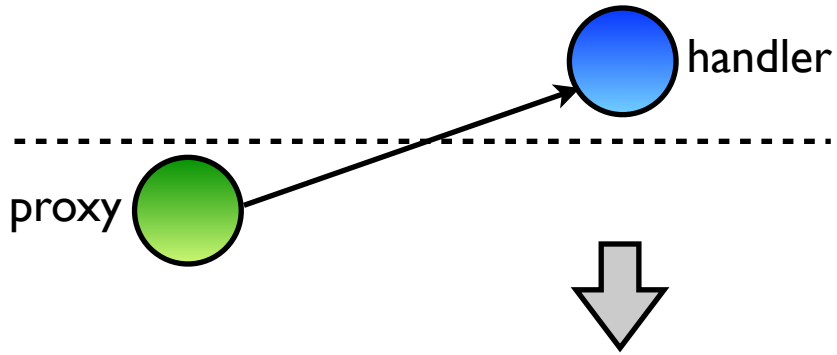
Proxies & frozen objects

- Frozen objects have strong invariants

- Proxies can emulate frozen objects, but handlers can't violate these invariants

```
var target = { x: 0 };
Object.freeze(target); // now target.x should be immutable

var y = 0;
var proxy = new Proxy(target, {
  get: function(tgt, name, rcvr) {
    return ++y;
  }
});

Object.isFrozen(proxy) // true!
proxy.x // error: cannot report inconsistent value for 'x'
```

Meta-level shifting




```
Object.getOwnPropertyDescriptor(proxy,name)
Object.defineProperty(proxy,name,pd)
Object.getOwnPropertyNames(proxy)
delete proxy.name
for (name in proxy) { ... }
for (name in Object.create(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)
```

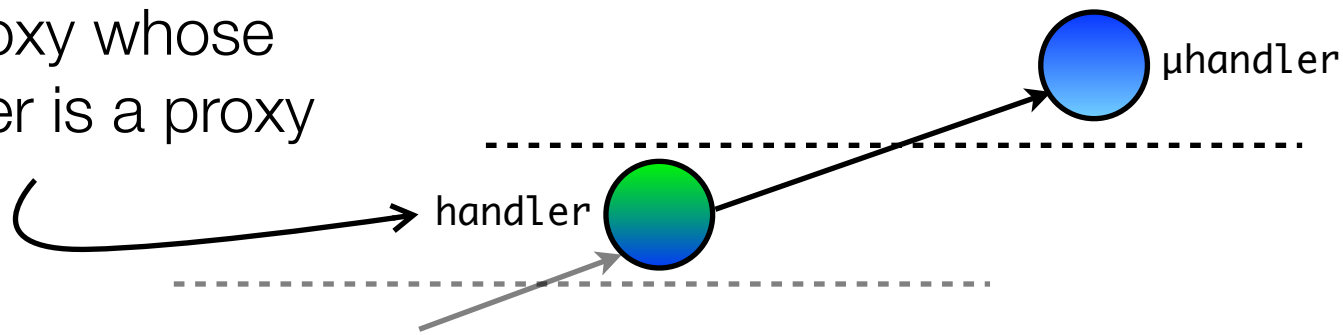
```
handler.getOwnPropertyDescriptor(target,name)
handler.defineProperty(target,name,pd)
handler.getOwnPropertyNames(target)
handler.deleteProperty(target,name)
handler.iterate(target)
handler.enumerate(target)
handler.{freeze|seal|preventExtensions}(target)
handler.has(target,name)
handler.hasOwn(target,name)
handler.keys(target)
handler.get(target,name,receiver)
handler.set(target,name,value,receiver)
handler.apply(target,receiver,args)
handler.construct(target,args)
```

base-level: many operations on objects

meta-level: all operations reified as invocations of traps

Meta-level shifting

a proxy whose handler is a proxy




```
handler.getOwnPropertyDescriptor(target,name)
handler.defineProperty(target,name,pd)
handler.getOwnPropertyNames(target)
handler.deleteProperty(target,name)
handler.iterate(target)
handler.enumerate(target)
handler.{freeze|seal|preventExtensions}(target)
handler.has(target,name)
handler.hasOwn(target,name)
handler.keys(target)
handler.get(target,name,rcvr)
handler.set(target,name,value,rcvr)
handler.apply(target,rcvr,args)
handler.construct(target,args)
```

meta-level: all operations reified as invocations of traps

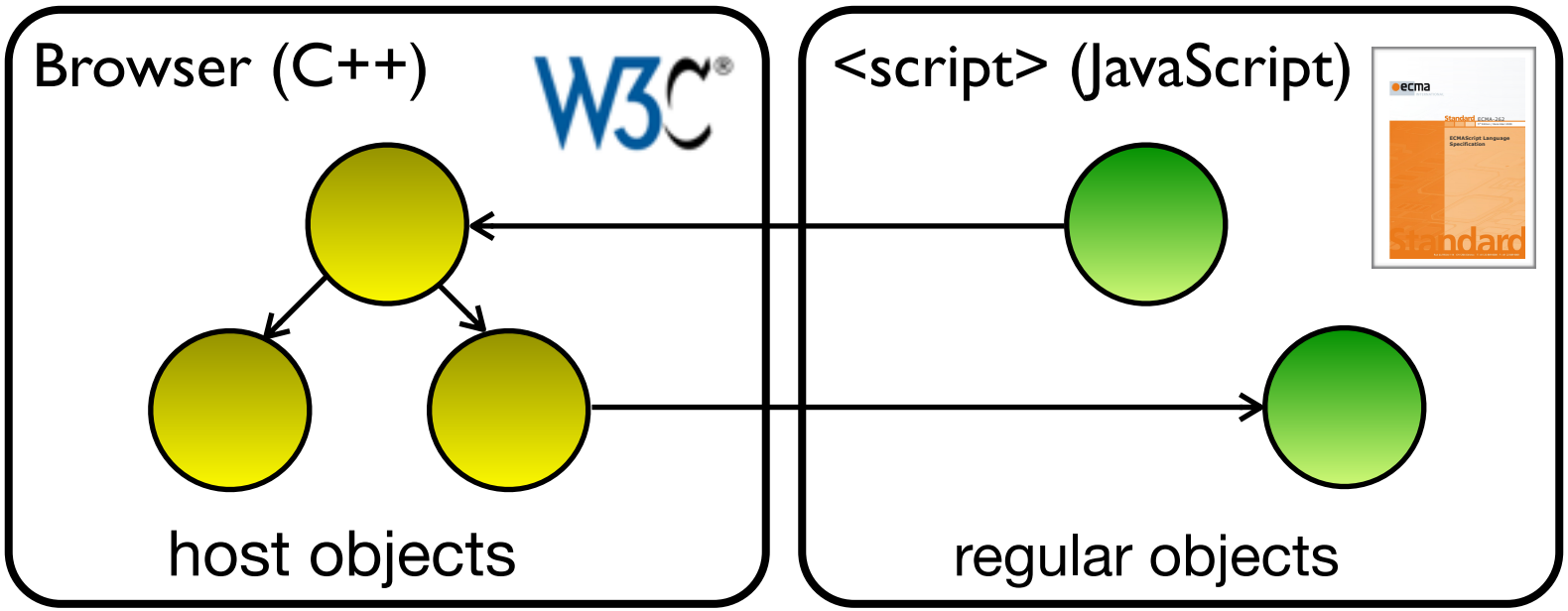
```
μhandler.get(tgt,‘getOwnPr..’)(target,name)
μhandler.get(tgt,‘definePr..’)(target,name,pd)
μhandler.get(tgt,‘getOwnPr..’)(target)
μhandler.get(tgt,‘deletePr..’)(target,name)
μhandler.get(tgt,‘iterate’)(target)
μhandler.get(tgt,‘enumerate’)(target)
μhandler.get(tgt,‘freeze’|..)(target)
μhandler.get(tgt,‘has’)(target,name)
μhandler.get(tgt,‘hasOwn’)(target,name)
μhandler.get(tgt,‘keys’)(target)
μhandler.get(tgt,‘get’)(target,name,rcvr)
μhandler.get(tgt,‘set’)(target,name,value,rcvr)
μhandler.get(tgt,‘apply’)(target,rcvr,args)
μhandler.get(tgt,‘construct’)(target,args)
```

meta-meta-level: all operations reified as invocations of ‘get’ trap

Applications of Proxies

Host objects

- Objects provided by the host platform
- E.g. the **DOM**: an object tree representation of the HTML document
- Appear to be Javascript objects, but not implemented in Javascript

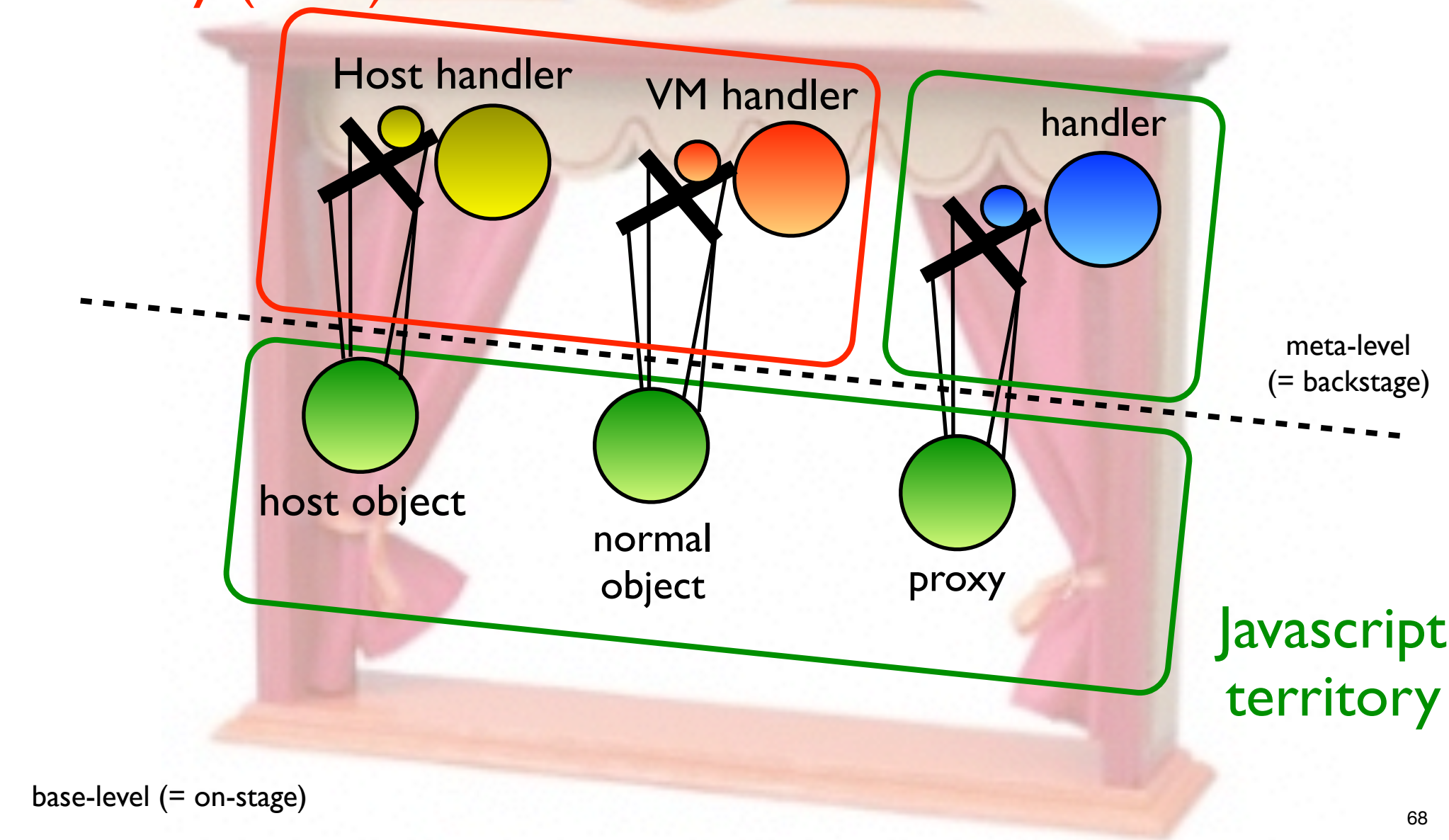


Host objects

- Can have odd behavior that regular JavaScript objects cannot emulate

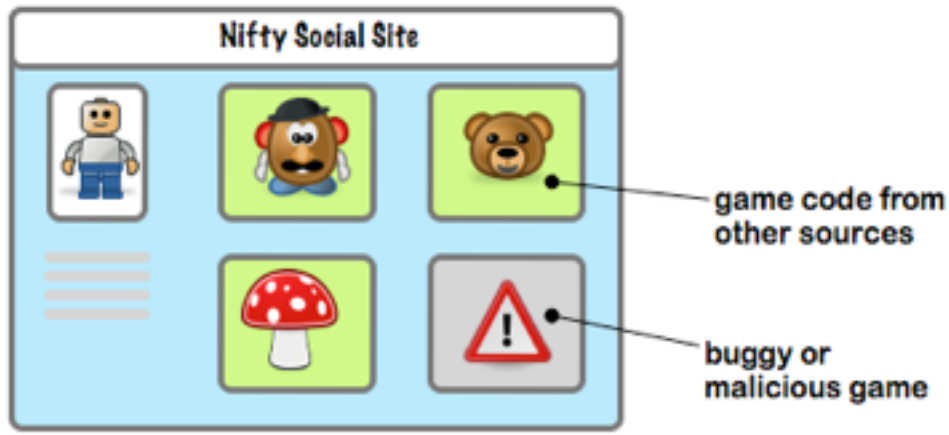
```
var links = document.getElementsByTagName('a');

links.length // 2
document.body.appendChild(newLink);
links.length // now 3
```

Retrofitting host objects



Caja

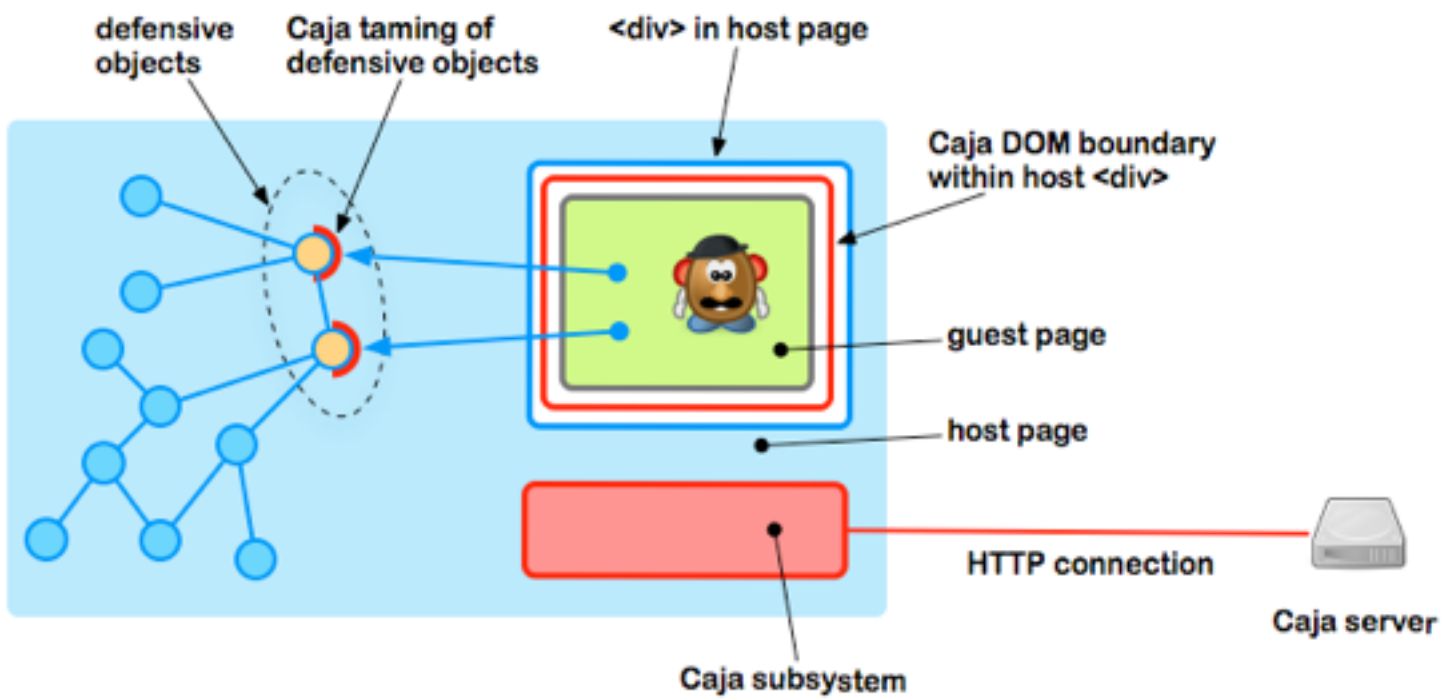
- Caja enables the safe embedding of third-party active content inside your website
  - Secures Google Sites, Google Apps Scripts, Google Earth Engine
- More generally: Gadgets, Mashups:



https://developers.google.com/caja/docs/about/

Caja : Taming

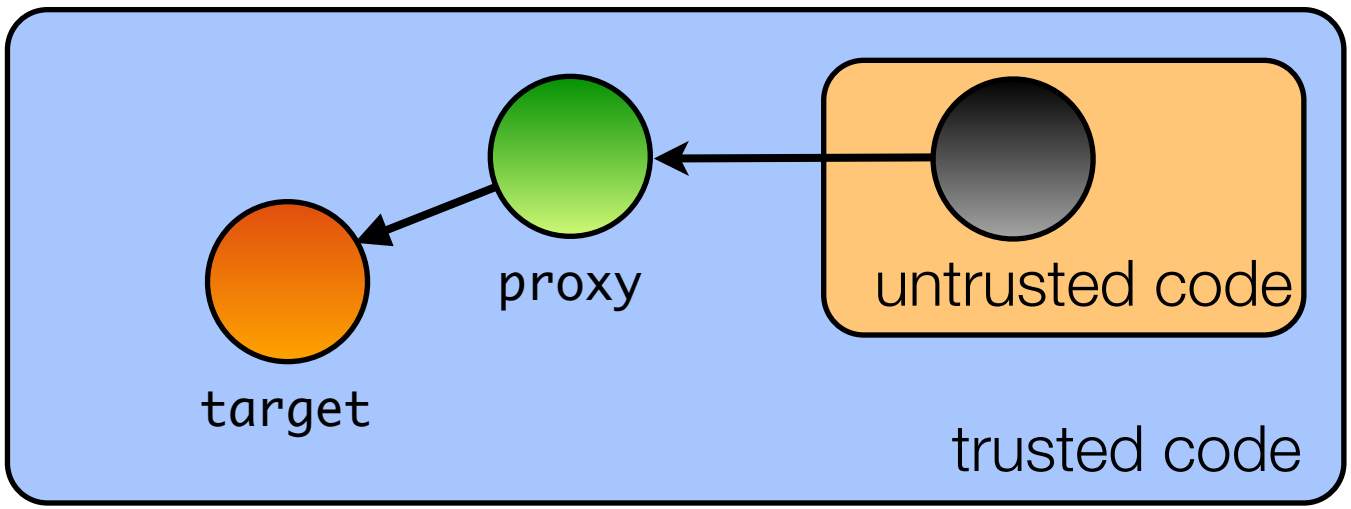
- Caja proxies the DOM. Untrusted content interacts with a virtual DOM, never with the real DOM.



https://developers.google.com/caja/docs/about/

Proxies again

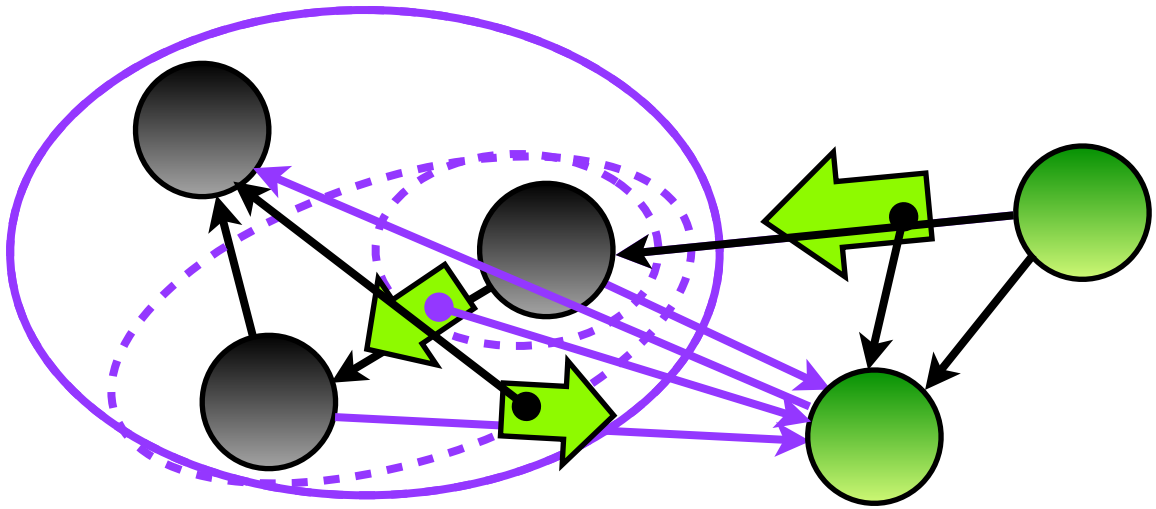
- Caja uses object capabilities to express security policies
- In the object-capability paradigm, an object is powerless unless given a reference to other (more) powerful objects
- Common to wrap objects with proxies that define a security policy
  - E.g. revocable reference: limit the lifetime of an object reference



Example: membranes

- Revocable references do not prevent permanent access to new references exposed through them.

- Membranes are *transitive* revocable references

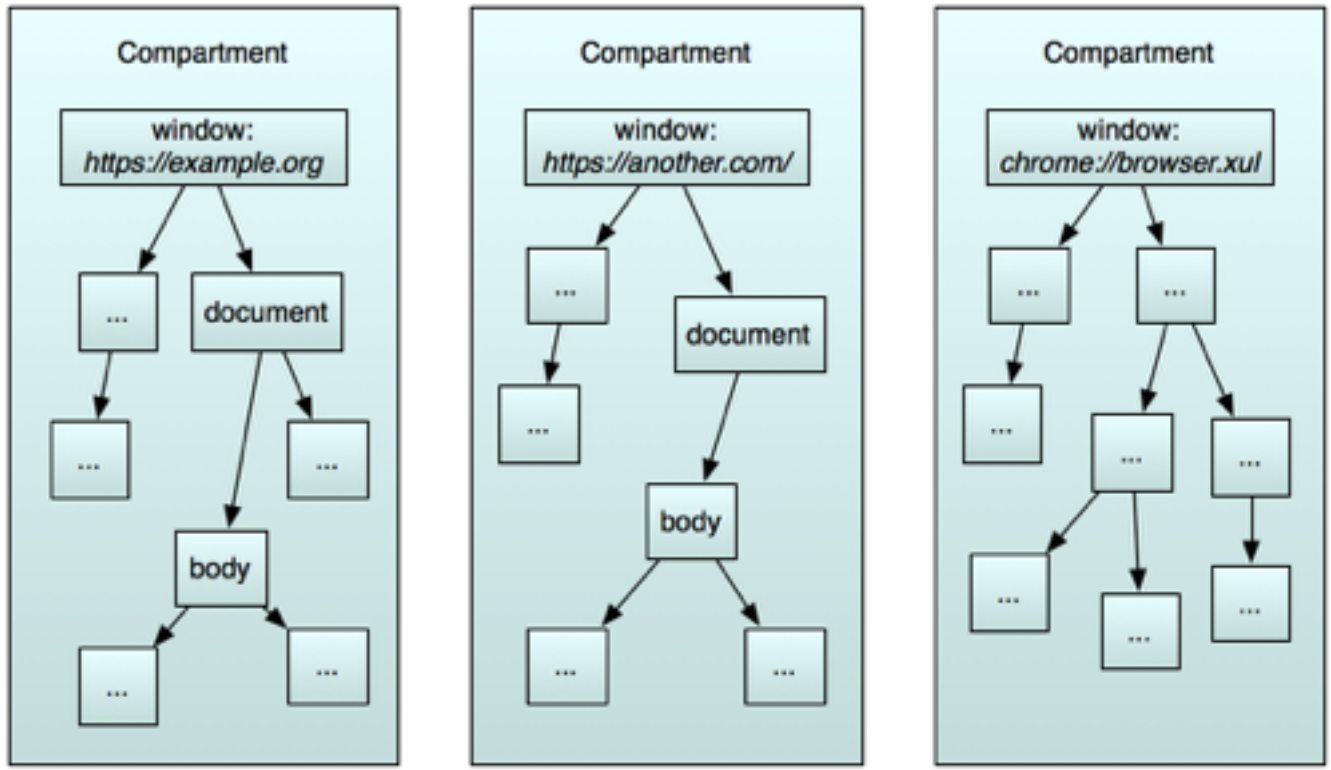


Example: membranes

```
function makeMembrane(initTarget) {
  var enabled = true;
  function wrap(target) {
    if (isPrimitive(target)) { return target; }
    var metaHandler = new Proxy(target, {
      get: function(target, trapName) {
        if (!enabled) { throw new Error("revoked"); }
        return function(...args) {
          return wrap(Reflect[trapName](...args.map(wrap)));
        }
      }
    });
    return new Proxy(target, metaHandler);
  }
  return {
    wrapper: wrap(initTarget),
    revoke: function() { enabled = false; }
  };
}
```

Gecko (Firefox)'s script security architecture

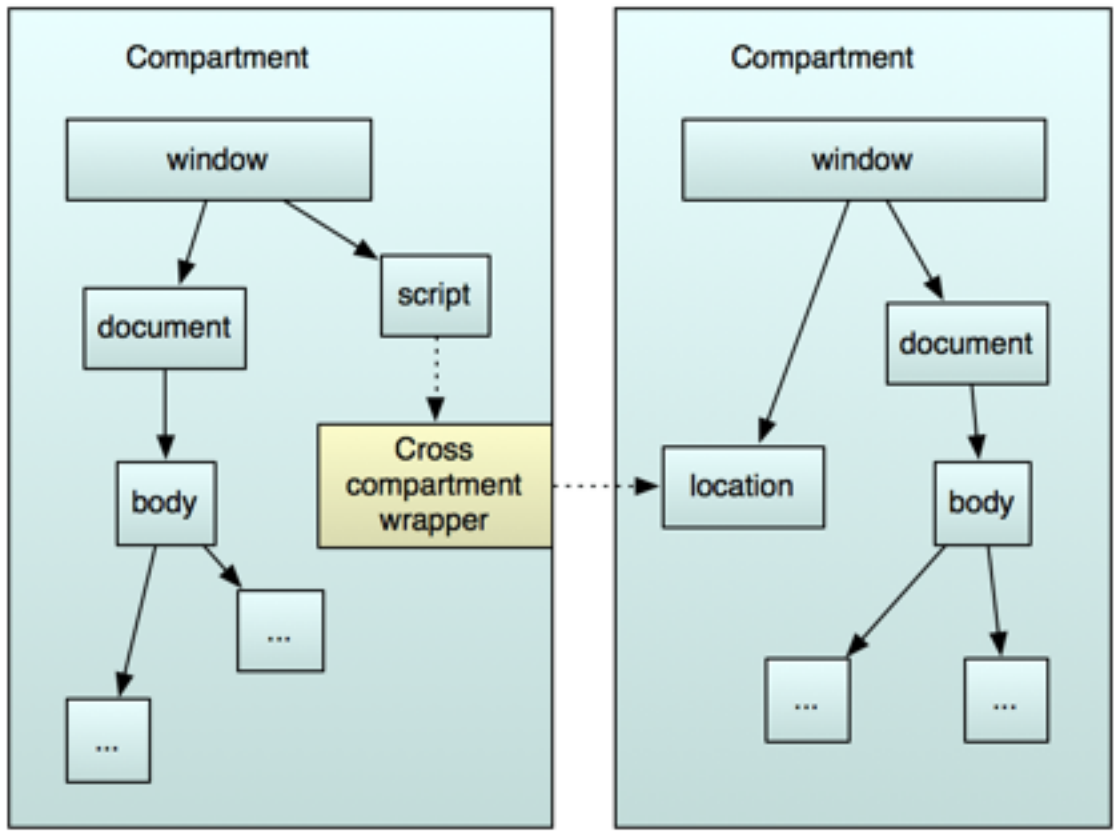
- Firefox uses proxies (membranes) internally to implement security boundaries between different site origins and privileged JS code. They form a key part of its security model.



https://developer.mozilla.org/en-US/docs/Mozilla/Gecko/Script_security

Gecko (Firefox)'s script security architecture

- Firefox uses proxies (membranes) internally to implement security boundaries between different site origins and privileged JS code. They form a key part of its security model.



https://developer.mozilla.org/en-US/docs/Mozilla/Gecko/Script_security

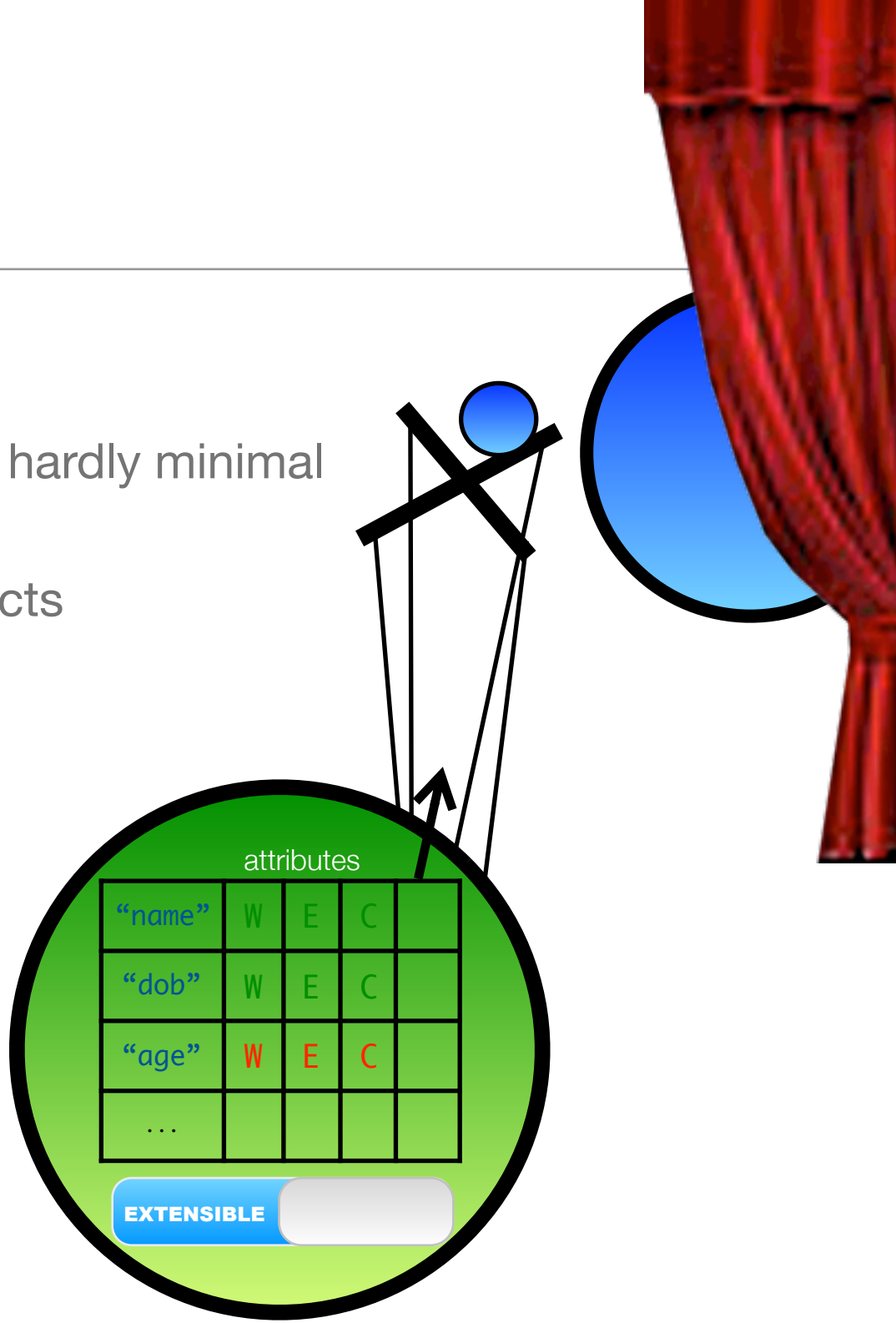
Gecko (Firefox)’s script security architecture

- Membranes helped avoid multiple high-severity security bugs

- Bobby Holley (Mozilla Engineer):
  “On the Right Fix, and Why the Bugzilla Breach Made Me Proud”
  http://bholley.net/blog/2016/the-right-fix.html

Conclusion

- Virtualization is a powerful idea
- Javascript: dynamic, flexible, but hardly minimal
- Proxies virtualize JavaScript objects
- Enables transparent sandboxing



References

- Warmly recommended: Doug Crockford on JavaScript http://goo.gl/FGxmM (YouTube playlist)