JavaScript security
or
JavaScript: the Good, the Bad, the Strict and the Secure Parts

Tom Van Cutsem

Software Languages Lab

Vrije Universiteit Brussel

Talk Outline

- Part I: JavaScript, the Good and the Bad parts
- Part II: ECMAScript 5 and Strict Mode
- Part III: ECMAScript 6 Proxies
- Part IV: Caja and Secure ECMAScript (SES)

Talk Outline

- This talk is about:
    - The JavaScript language proper
    - Language dialects and features to enable or improve security
- This talk is not about:
    - Security exploits in JavaScript, or how to avoid specific exploits (e.g. XSS attacks)

About Me

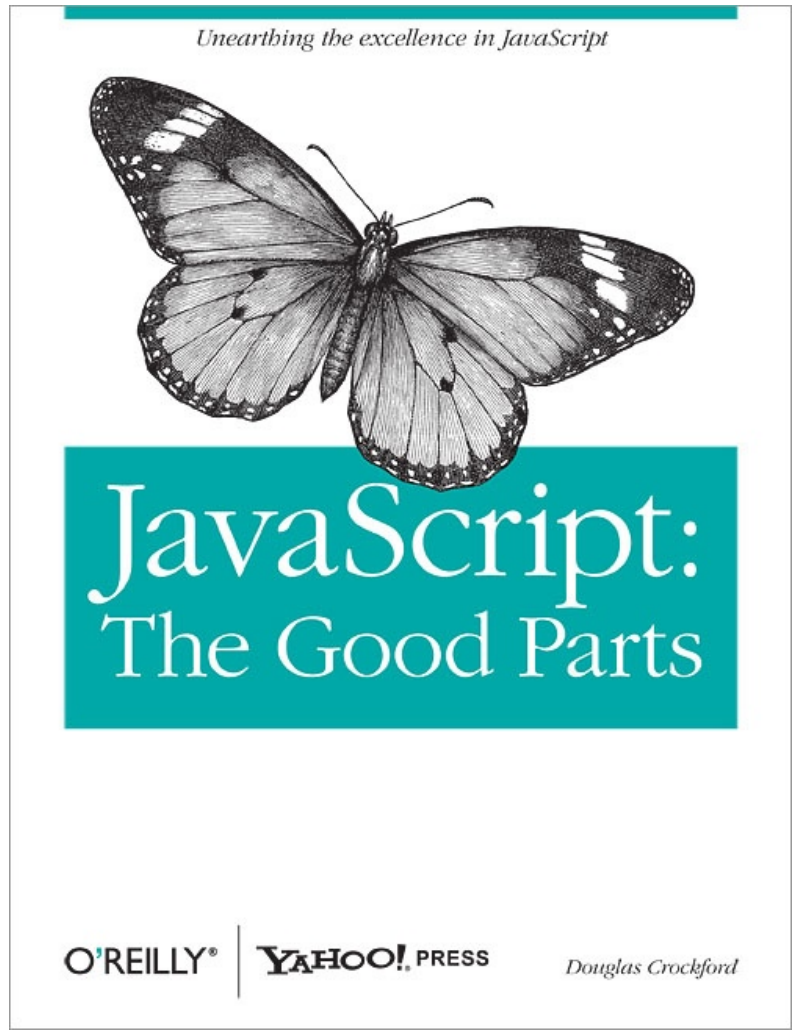
- Professor of Computer Science at Vrije Universiteit Brussel, Belgium
    - Programming Languages, concurrent and distributed programming
- ECMA TC39 (Javascript standardization committee)
- Visiting Faculty at the Google Caja team (2010)

Part I: Javascript, the Good and the Bad parts

JavaScript

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

The world's most misunderstood language



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

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...

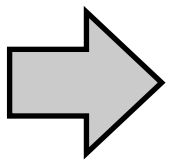
```
// computed property access and assignment
obj[“foo”]
obj[“foo”] = 42;

// 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;
```

Bad Parts: global variables

- Scripts depend on global variables for linkage

Bad

```
<script>
var x; // global
var lib = {...};
</script>


<script>
... lib ...
</script>
```



Better

```
<script>
var lib = (function(){
  var x; // local
  return {...};
}())
</script>
```

Bad Parts: with statement

- `with`-statement breaks static scoping

```
with (expr) {
   ... x ...
}
```

```
var x = 42;
var obj = {};
with (obj) {
   print(x); // 42
   obj.x = 24;
   print(x); // 24
}
```

More Bad Parts

- “var hoisting”: variables are not block-scoped but function-scoped
- Implicit type coercions
- No integers (all numbers are IEEE 754 floating point)
- Automatic semicolon insertion
- ...

Delving Deeper

- Some finer points about
    - Functions
    - Objects
    - Methods

Functions

- Functions are objects

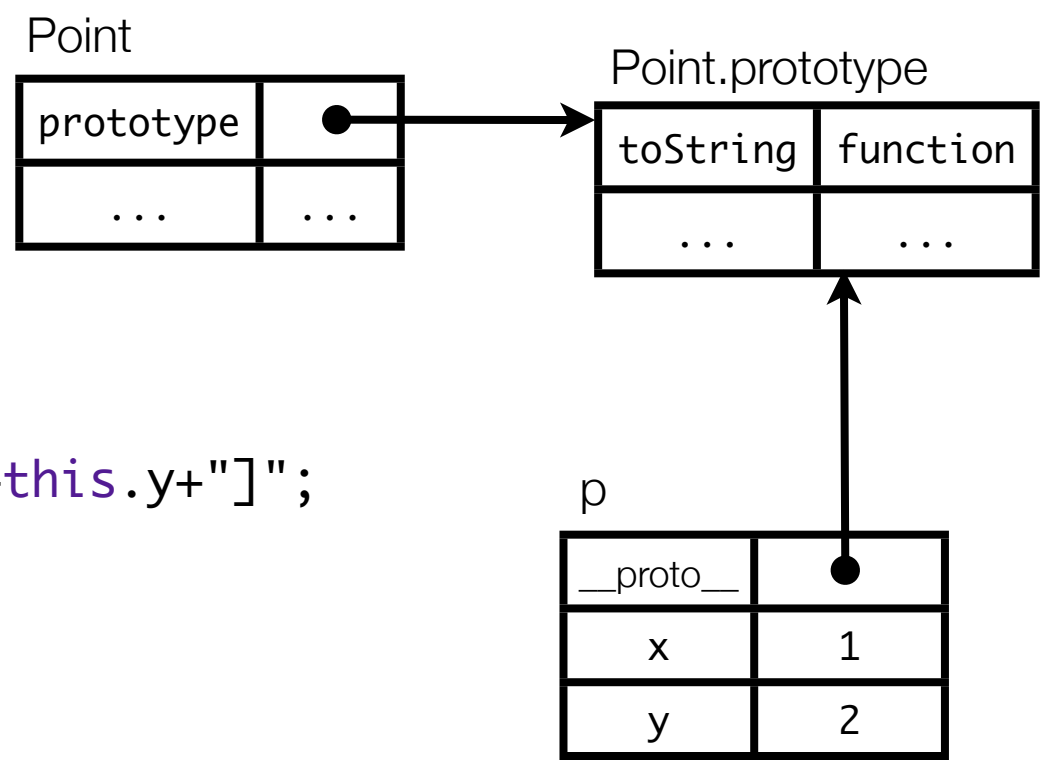
```
function add(x,y) { return x + y; }
add(1,2) // 3

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

Objects

- No classes.
- Functions may act as object constructors.
- All objects have a “prototype”: object-based inheritance

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);
```



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); // new in ES5

indexf(0) // 10
```

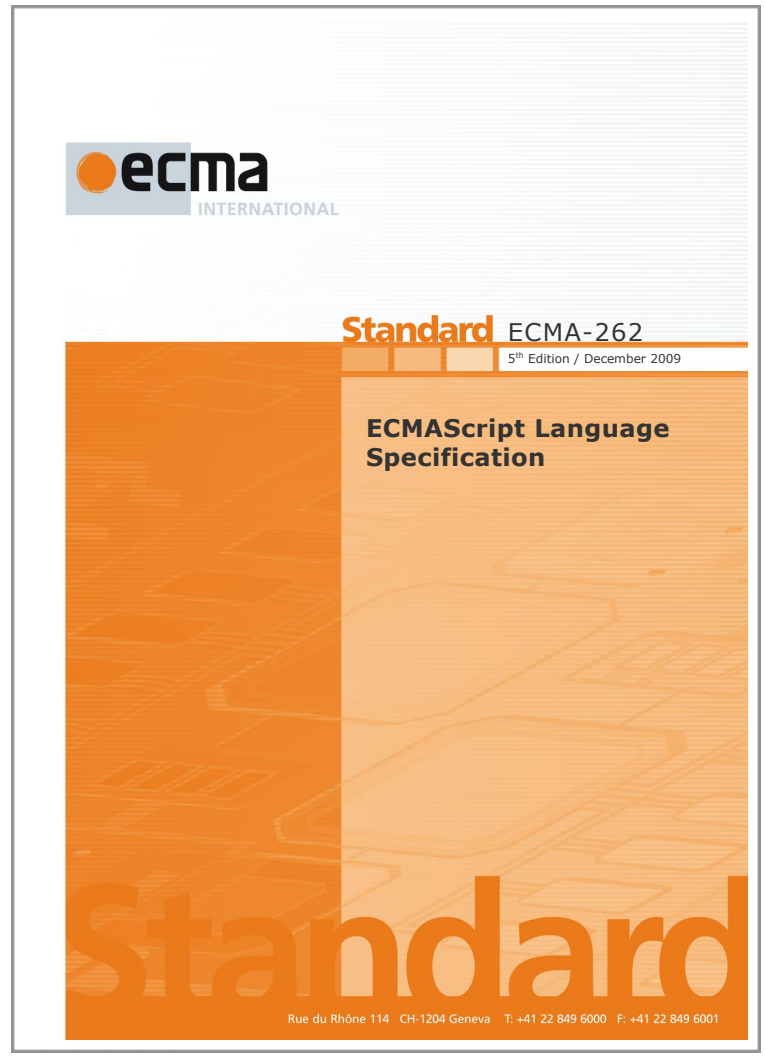
Summary so far

- Javascript: “a Lisp in C’s clothing” (D. Crockford)
- Good parts: functions, object literals
- Bad parts: global vars, lack of static scoping, var hoisting
- No way to protect an object from modifications by its clients
    - Unsafe to share objects across trust boundaries

Part II: ECMAScript 5 and Strict Mode

ECMAScript

- “Standard” Javascript
  - 1st ed. 1997
  - 2nd ed. 1998
  - 3rd ed. 1999
  - ~~4th ed.~~
  - 5th ed. 2009
  - *6th ed. end of 2013 (tentative)*



ECMAScript 5 Themes

- New APIs
    - Array methods, JSON, bound functions, ...
- More robust programming
    - Tamper-proof objects, strict mode, ...
- Better emulation of host objects (e.g. the DOM)
    - Accessors (getter / setter properties), property attributes

ECMAScript 5 Themes

- **New APIs**
  - Array methods, JSON, bound functions, ...
- More robust programming
  - Tamper-proof objects, strict mode, ...
- Better emulation of host objects (e.g. the DOM)
  - Accessors (getter / setter properties), property attributes

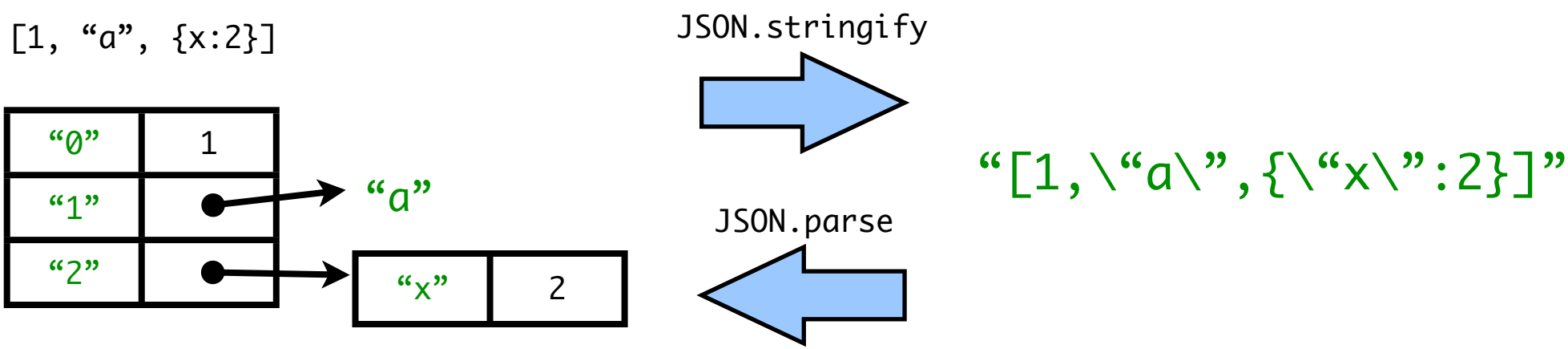
JSON

- **J**ava**S**cript **O**bject **N**otation
- A simple subset of Javascript (numbers, strings, arrays and objects without methods)
- Formal syntax literally fits *in a margin*. See http://json.org/

```
{ “name” : “Bob”,
  “age” : 42,
  “address” : {
    “street” : “main st”
  }
}
```

ECMAScript 5 and JSON

- Before ES5, could either parse quickly or safely
- Unsafe: `eval(jsonString)`
- In ES5: use JSON.parse, JSON.stringify



ECMAScript 5 Themes

- New APIs
    - Array methods, JSON, bound functions, ...
- **More robust programming**
    - Tamper-proof objects, strict mode, ...
- Better emulation of host objects (e.g. the DOM)
    - Accessors (getter / setter properties), property attributes

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
```

Ecmascript 5 Strict mode

- Safer subset of the language
- No silent errors
- True static scoping rules
- No global object leakage

Ecmascript 5 Strict mode

- Explicit opt-in to avoid backwards compatibility constraints
- How to opt-in
  - Per “program” (file, script tag, ...)
  - Per function
- Strict and non-strict mode code can interact (e.g. on the same web page)

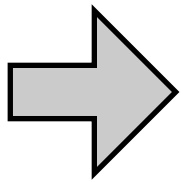
```
<script>
"use strict";
...
</script>
```

```
function f() {
  "use strict";
  ...
}
```

Strict-mode opt-in: gotcha's

- Beware: minification and deployment tools may concatenate scripts

```
<script>
"use strict";
// in strict mode
</script>
```



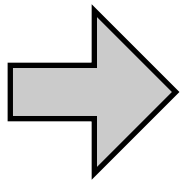
```
<script>
"use strict";
// in strict mode
<script>
// not in strict mode
function f(){...}
</script>
```

```
// f is now
// accidentally strict!
function f(){...}
</script>
```

Strict-mode opt-in: gotcha's

- Suggested refactoring is to wrap script blocks in function bodies

```
<script>
(function(){
  "use strict";
  // in strict mode
}())
</script>
```



```
<script>
(function(){
  "use strict";
  // in strict mode
})
```

```
<script>
// not in strict mode
function f(){...}
</script>
```

```
// not in strict mode
function f(){...}
</script>
```

Static scoping in ES5

- ECMAScript 5 non-strict is not statically scoped
- Four violations:
  - Assigning to a non-existent variable creates a new global variable

```
function f() { var xfoo; xFoo = 1; }
```

  - `with (expr) { x }` statement
  - `delete x;` // may delete a statically visible var
  - `eval('var x=8');` // may add a statically visible var

Ecmascript 5 Strict: syntactic restrictions

- The following are forbidden in strict mode (signaled as syntax errors):

```
with (expr) {
  ...
}
```

```
var x = 5;
...
delete x; // deleting a var
```

```
{ a: 1,
  b: 2,
  b: 3 } // duplicate property
```

```
var n = 023; // octal literal
```

```
function f(a,b,b) {
  // repeated param name
}
```

```
function f(eval) {
  // eval as variable name
}
```

Ecmascript 5 Strict

- Runtime changes (fail silently outside of strict mode, throw an exception in strict mode)

```
function f() {
  // assigning to an undeclared variable
  var xfoo;
  xFoo = 1;
}
```

```
// deleting a non-configurable property
var pt = Object.freeze({x:0,y:0});
delete pt.x;
```

Ecmascript 5 Strict: avoid global object leakage

- Runtime changes: default `this` bound to `undefined` instead of the global object

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

var p = new Point(1,2);
var p = Point(1,2);
// window.x = 1;
// window.y = 2;
print(x) // 1
```

```
“use strict”;
function Point(x, y) {
  this.x = x;
  this.y = y;
}

var p = new Point(1,2);

var p = Point(1,2);
// undefined.x = 1;
// error!
```

Direct versus Indirect Eval

- ES5 runtime changes to eval (both in strict and non-strict mode)
- `eval` “operator” versus `eval` “function”

Direct

```
var x = 0;
eval(“x = 5”);
print(x); // 5
```

Indirect

```
var x = 0;
var f = eval;
f(“x = 5”);
print(x); // 0
```

ECMAScript 5 Themes

- New APIs
    - Array methods, JSON, bound functions, ...
- More robust programming
    - Tamper-proof objects, strict mode, ...
- **Better emulation of host objects** (e.g. the DOM)
    - Accessors (getter / setter properties), property attributes

Host objects

- Objects provided by the host platform
- E.g. the **DOM**: a tree representation of the HTML document
- “look and feel” like Javascript objects, but are not implemented in Javascript (typically in C++)
- Odd behavior not always easy to reproduce in JavaScript itself
    - ES5 provides APIs that partially solve this (accessors, property attributes)
    - ES6 goes further with proxies

Part III: ECMAScript 6 Proxies

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

Why Proxies?

- Generic access control wrappers:
  - E.g. revocable references, membranes (see later)
- Emulating host objects
  - Ability to self-host the DOM
  - Could provide a “virtual” document to third-party scripts

Generic wrapper example: tracing

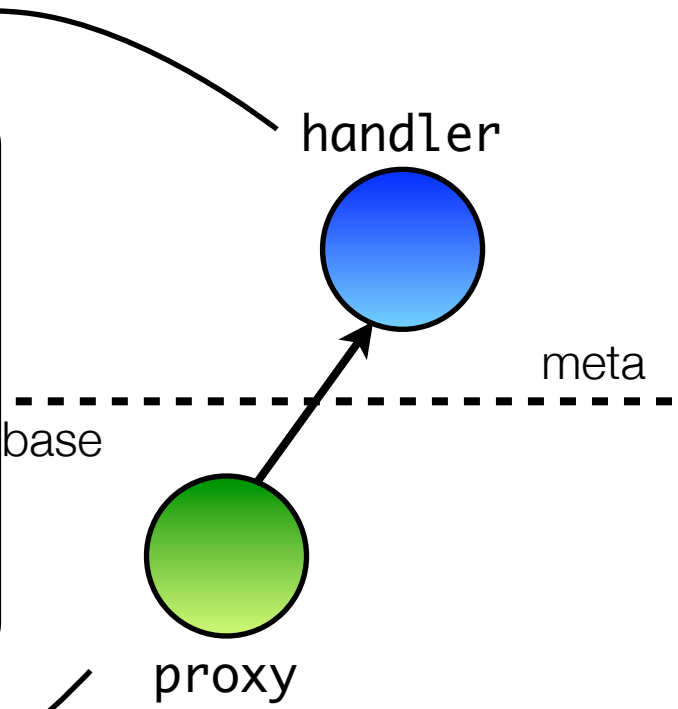
```
function makePoint(x, y) {
  return {
    x: x,
    y: y
  };
}


var p = makePoint(2,2);
var tp = makeTracer(p);
tp.x
// log(p, ‘get’, ‘x’);
// 2
tp.y = 3
// log(p, ‘set’, ‘y’, 3);
// 3
```

Generic wrapper example: tracing

```
function makeTracer(obj) {
  var proxy = Proxy(obj, {
    get: function(tgt, name) {
      console.log(tgt, ‘get’, name);
      return tgt[name];
    },
    set: function(tgt, name, val) {
      console.log(tgt, ‘set’, name, val);
      return tgt[name] = val;
    },
  });
  return proxy;
}
```

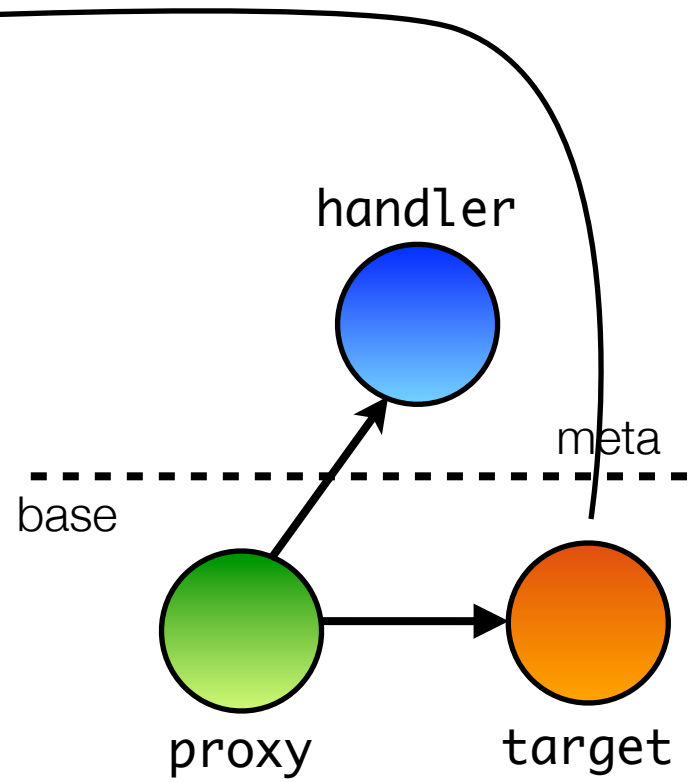
Generic wrapper example: tracing

```
function makeTracer(obj) {
  var proxy = Proxy(obj, {
    get: function(tgt, name) {
      console.log(tgt, 'get', name);
      return tgt[name];
    },
    set: function(tgt, name, val) {
      console.log(tgt, 'set', name, val);
      return tgt[name] = val;
    },
  });
  return proxy;
}
```



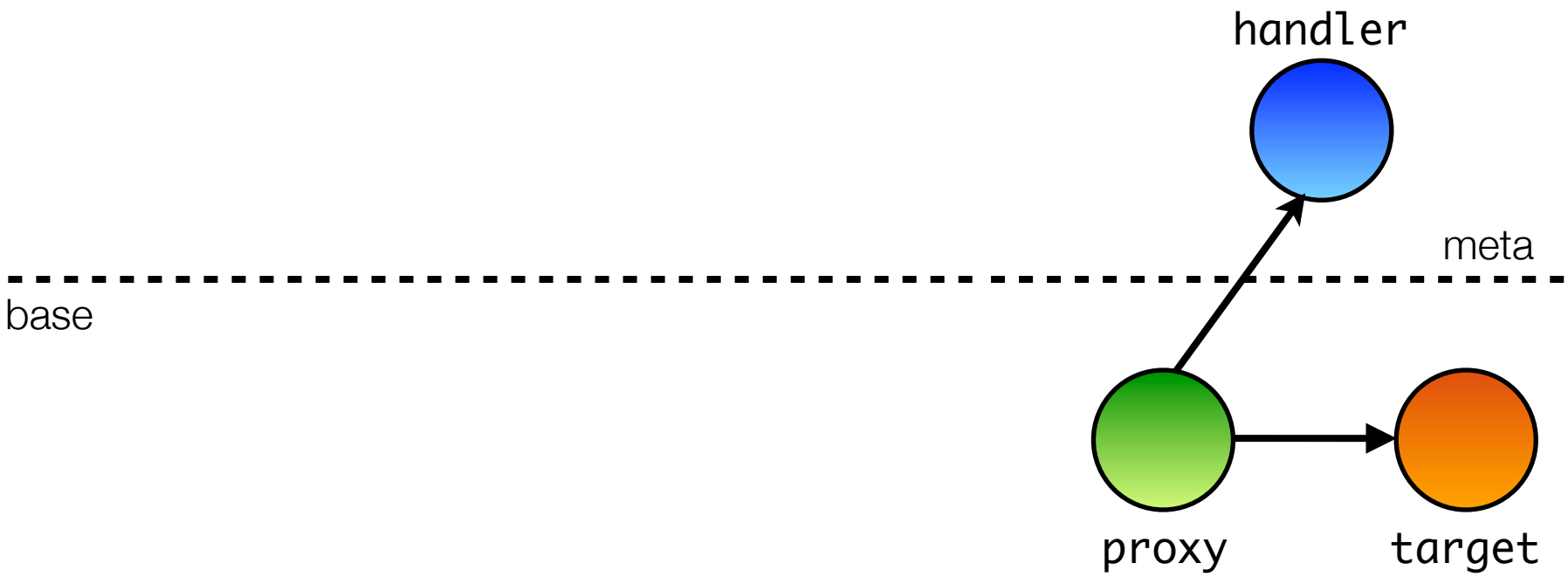
Generic wrapper example: tracing

```
function makeTracer(obj) {
  var proxy = Proxy(obj, {
    get: function(tgt, name) {
      console.log(tgt, 'get', name);
      return tgt[name];
    },
    set: function(tgt, name, val) {
      console.log(tgt, 'set', name, val);
      return tgt[name] = val;
    },
  });
  return proxy;
}
```



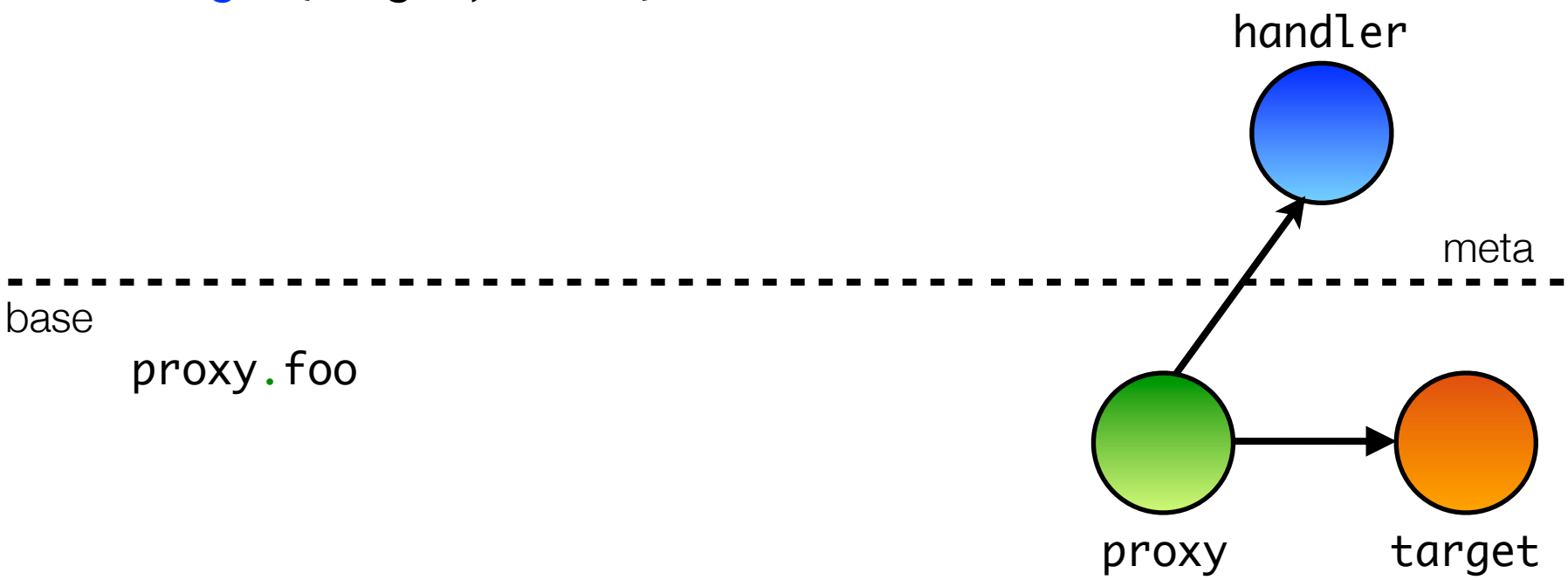
Stratified API

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



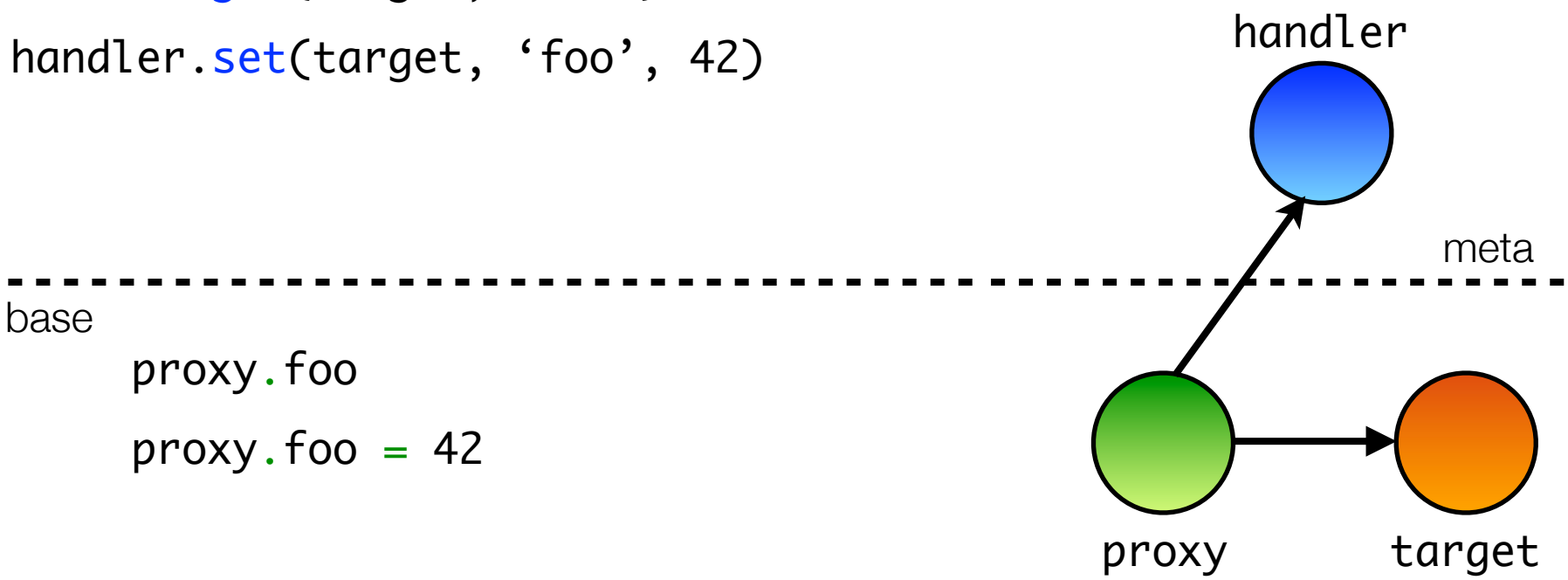
Stratified API

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


handler.get(target, 'foo')
```



Stratified API

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

handler.get(target, 'foo')
handler.set(target, 'foo', 42)
```



Stratified API

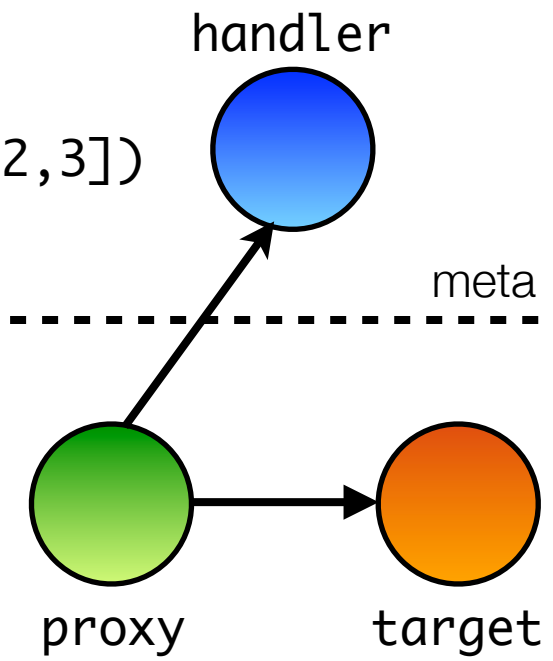
```
var proxy = Proxy(target, handler);
```

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

meta

base

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



Stratified API

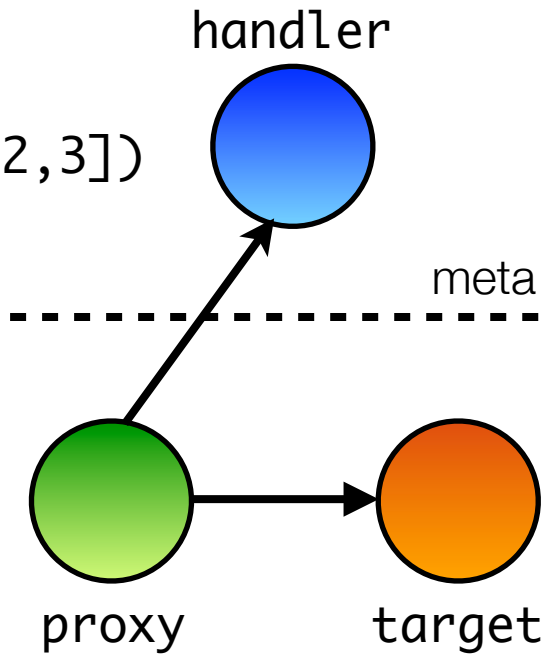
```
var proxy = Proxy(target, handler);
```

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

meta

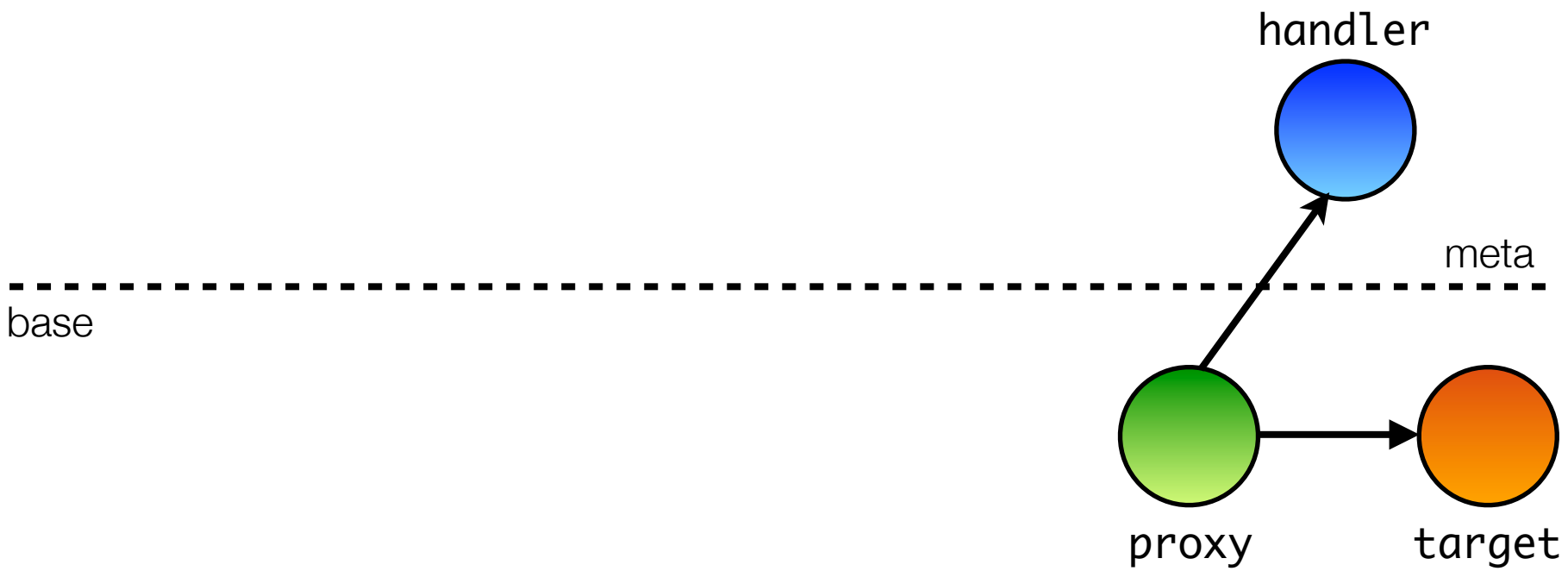
base

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



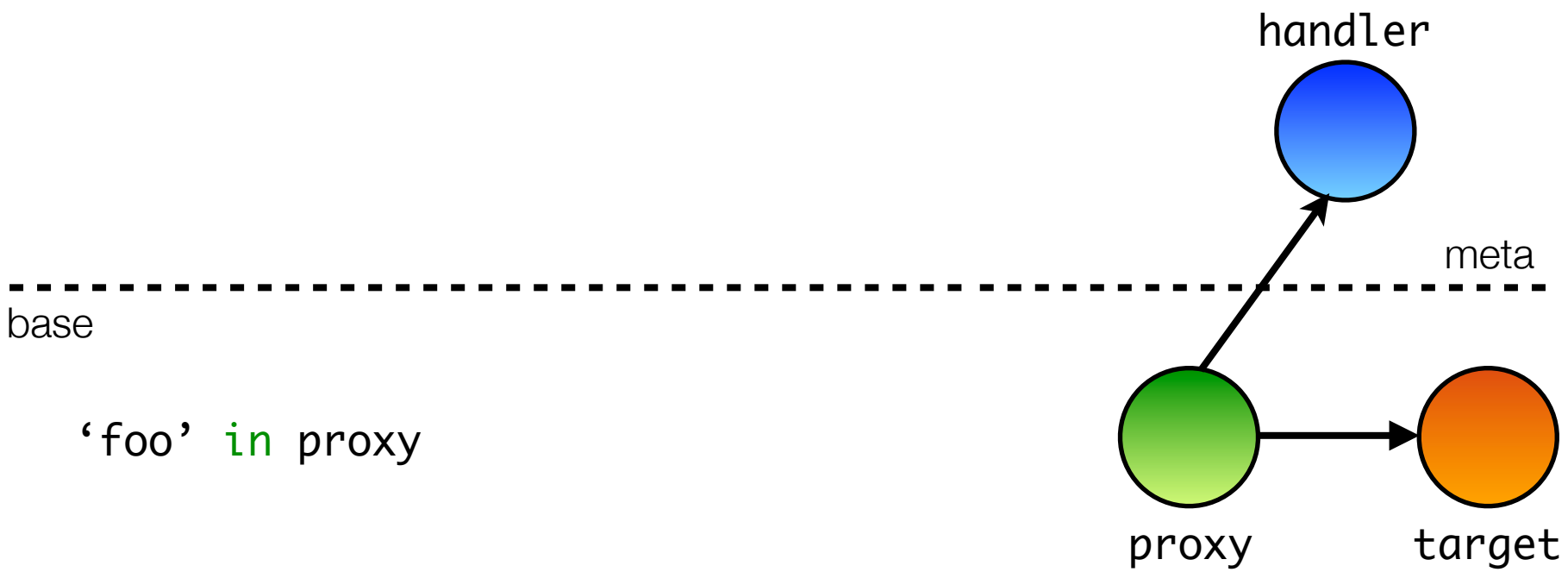
Not just property access

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



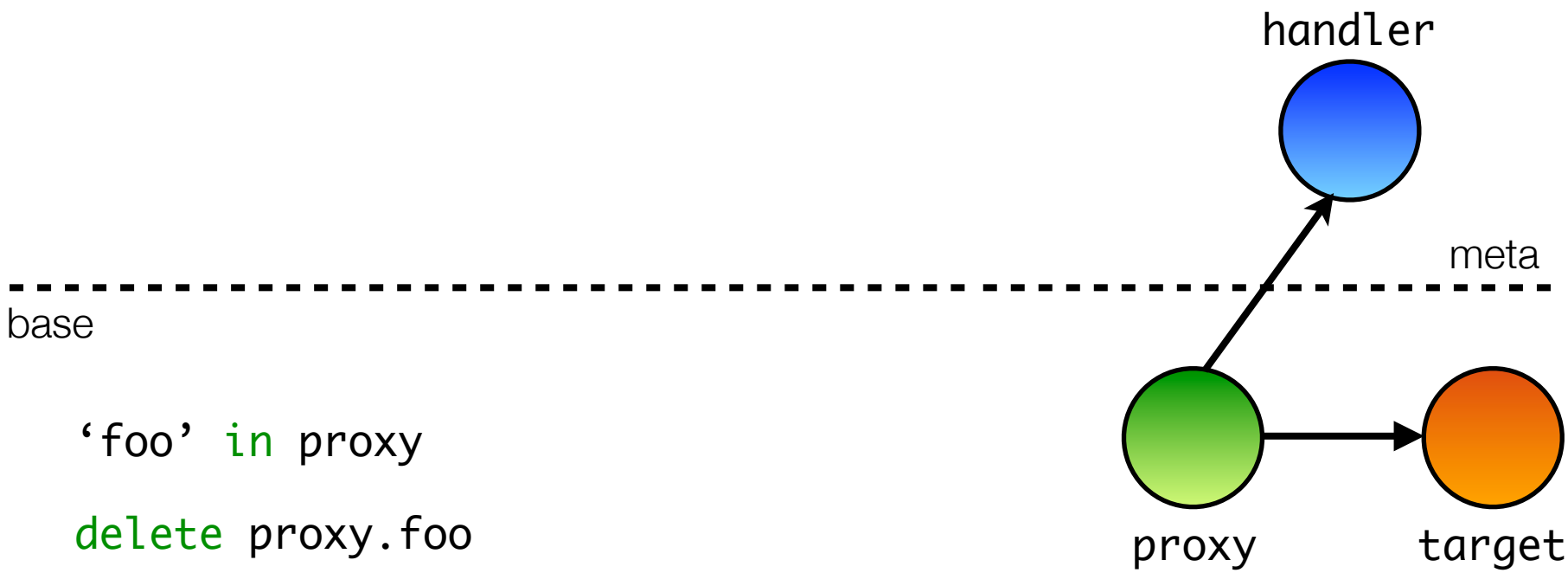
Not just property access

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

handler.has(target,'foo')
```



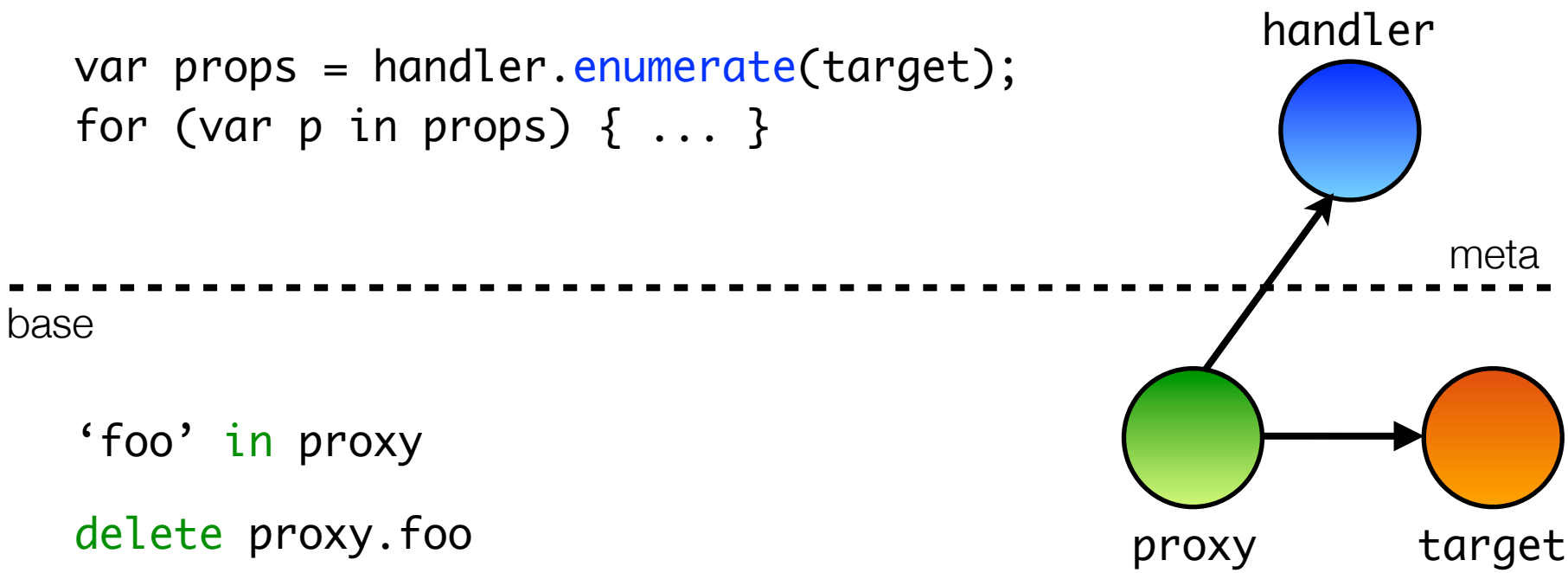
Not just property access

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

handler.has(target,‘foo’)

handler.deleteProperty(target,‘foo’)
```



Not just property access

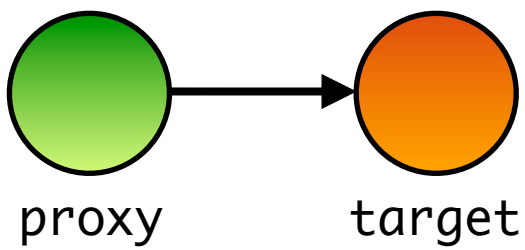
```
var proxy = Proxy(target, handler);

handler.has(target,‘foo’)

handler.deleteProperty(target,‘foo’)

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




```
‘foo’ in proxy

delete proxy.foo

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

Example: a revocable reference

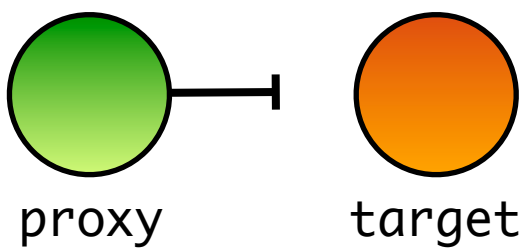
- revocable reference: limit the lifetime of an object reference




```
var ref = makeRevocable(target);
var proxy = ref.proxy;
// pass proxy to code
ref.revoke();
```

Example: a revocable reference

- revocable reference: limit the lifetime of an object reference




```
var ref = makeRevocable(target);
var proxy = ref.proxy;
// pass proxy to code
ref.revoke();
```

Example: a revocable reference

```
function makeRevocable(target) {
  var enabled = true;
  return {
    proxy: Proxy(target, {
      get: function(target, name) {
        if (enabled) { return target[name]; }
        throw new Error(“revoked”);
      }
    }),
    revoke: function() { enabled = false; };
  }
}
```

Proxies: availability

- Firefox
- `node --harmony`
- Chrome (enable experimental JS flag in chrome://flags)
- Library that implements the latest API proposed for ES6

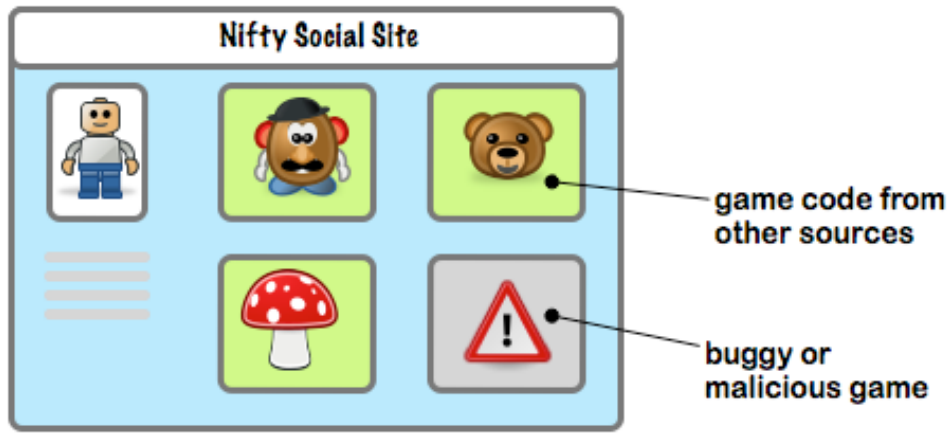
```
<script src="reflect.js"></script>
```

- Available on Github: https://github.com/tvcutsem/harmony-reflect

Part IV: Caja and Secure ECMAScript (SES)

Caja

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



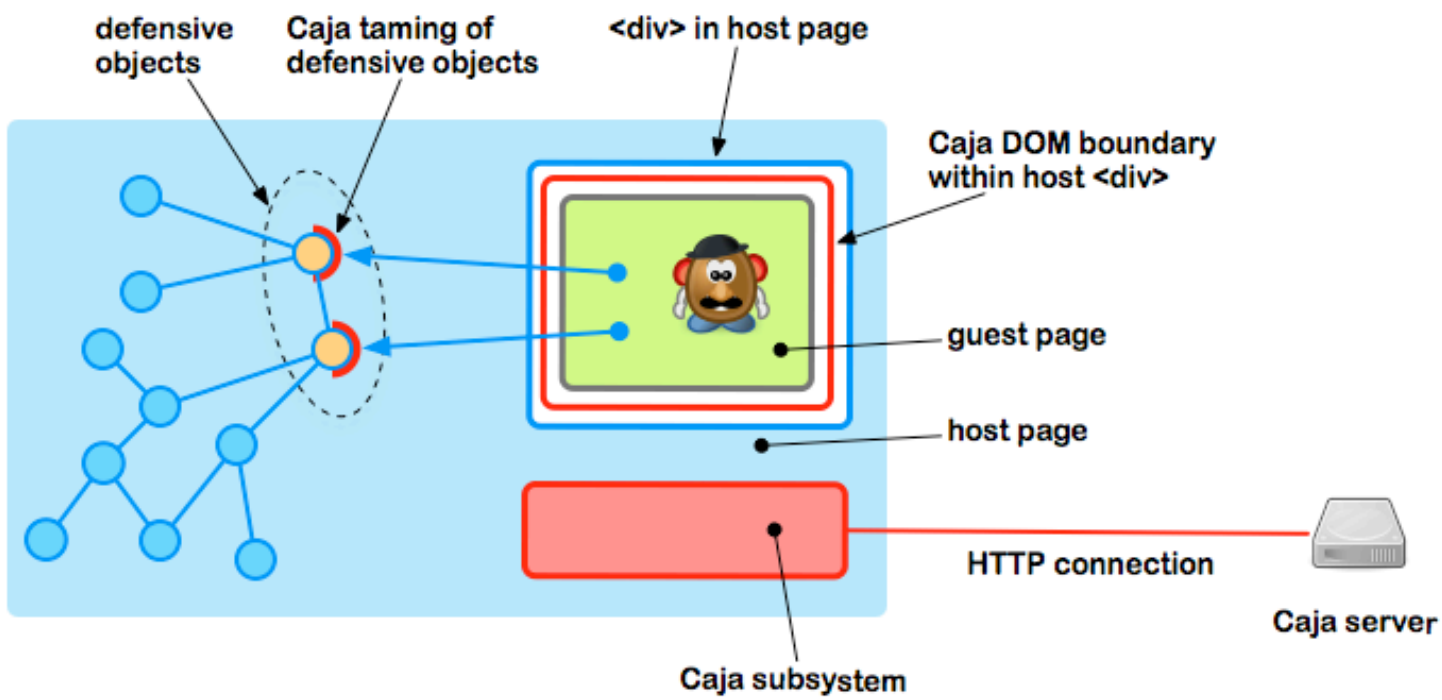
https://developers.google.com/caja/docs/about/

Caja 

- Not a traditional sandbox. Caja-compiled code is safe to inline directly in a webpage . No iframes.
- Can put multiple third-party apps into the same page and allow them to directly exchange JavaScript objects
  - Great for writing mash-ups
- The host page is protected from the embedded apps
  - E.g. embedded app can't redirect the host page to phishing sites, or steal cookies from the hosting page

Caja : Taming

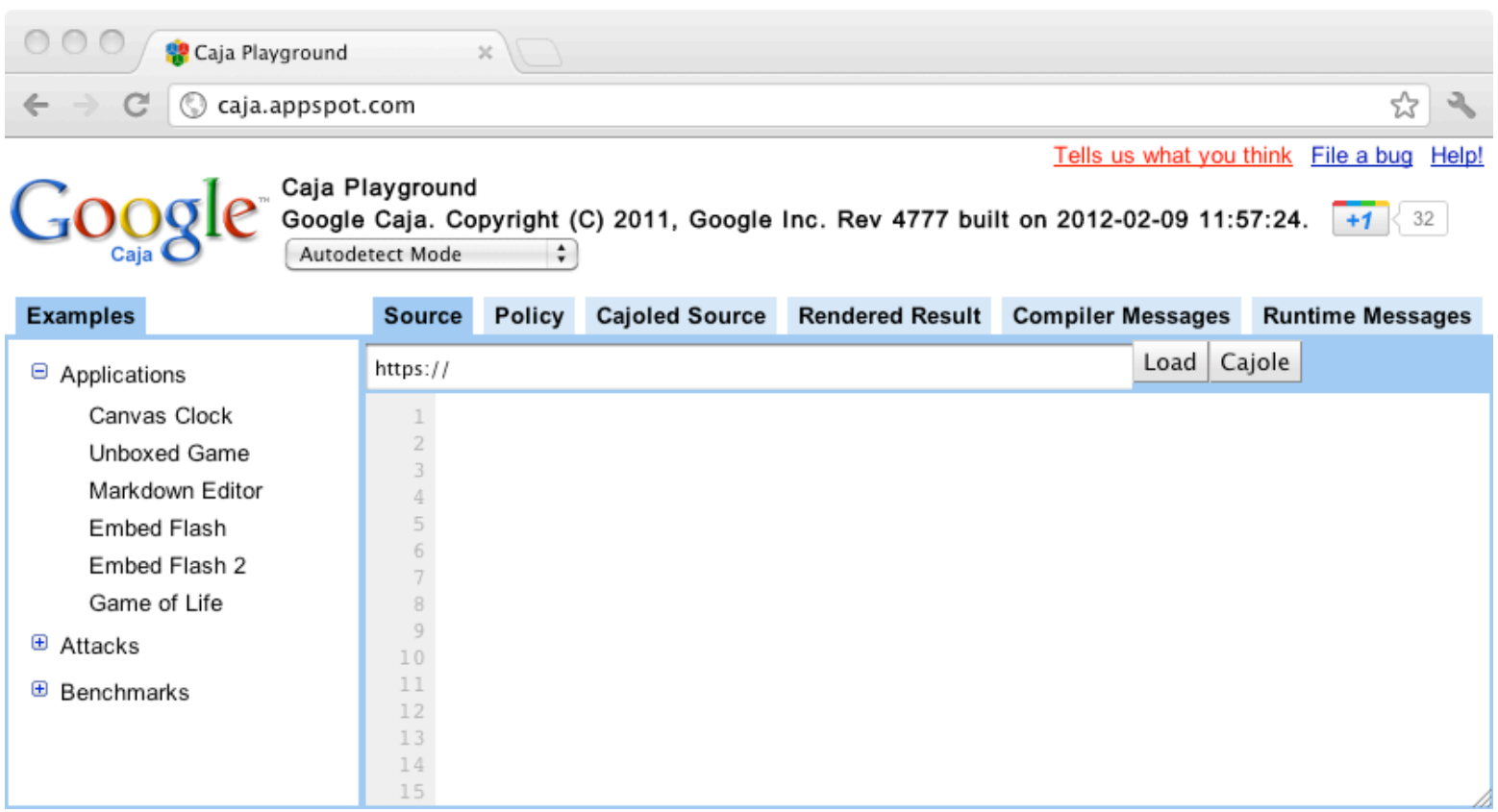
- Caja proxies the DOM. Untrusted content interacts with a virtual DOM, never with the real DOM.



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

Caja

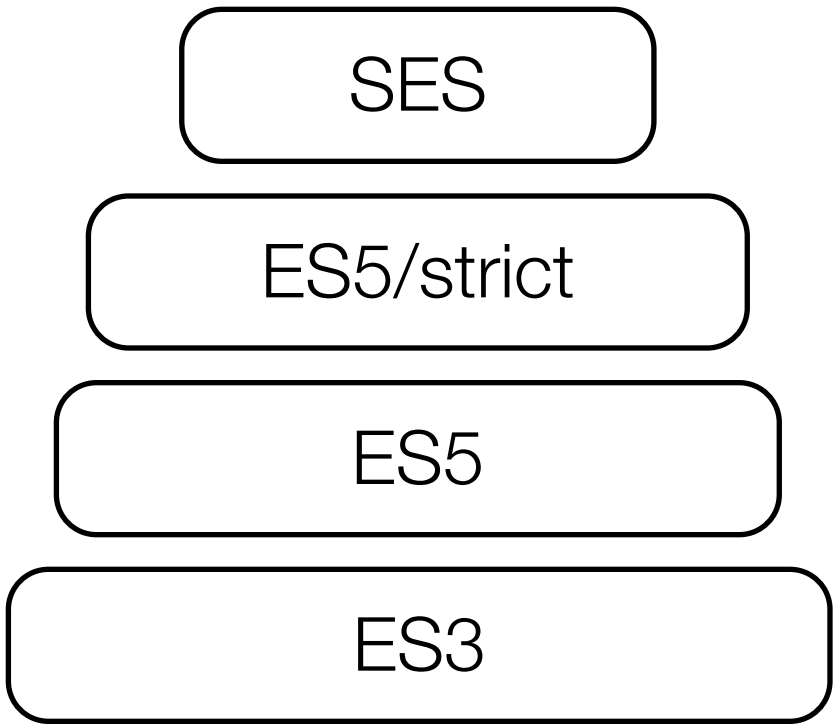
- Example: Caja Playground
- http://caja.appspot.com



Caja 

- Caja consists of:
    - A capability-secure JavaScript subset (SES)
    - A safe DOM wrapper (Domado)
    - A HTML and CSS sanitizer
- SES is the portion of Caja responsible for securing JavaScript

Secure ECMAScript



adds confinement

adds proper static scoping

adds tamper-proof objects

Secure ECMAScript

- Implemented as a library on top of ES5/strict
- Include as first script, before any other JavaScript code runs:

```
<script src="startSES.js"></script>
```

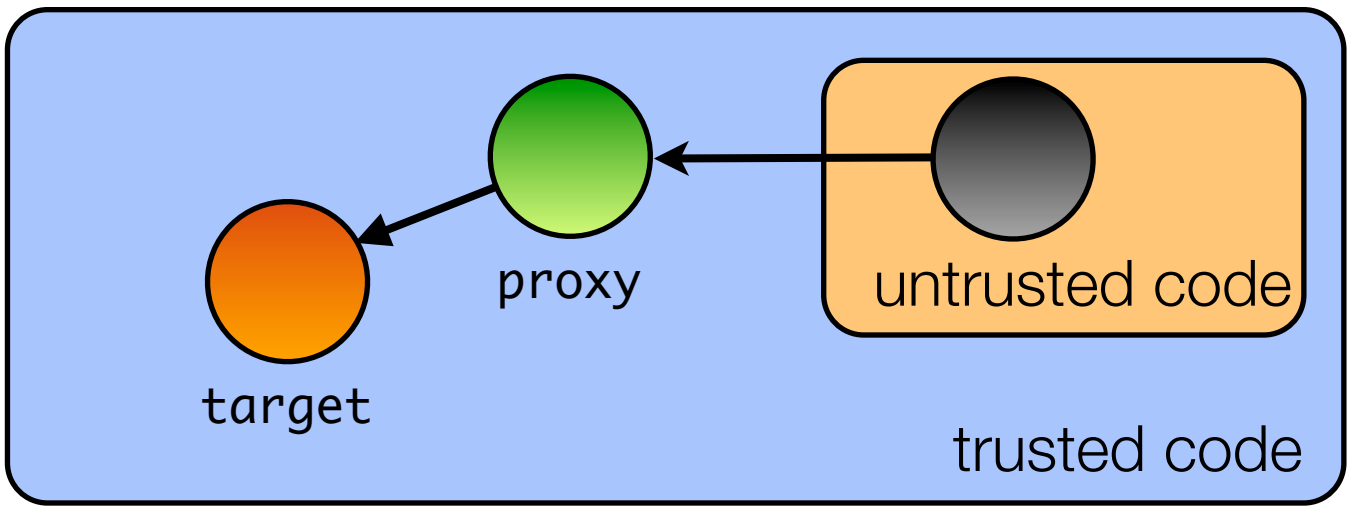
Secure ECMAScript

```
<script src="startSES.js"></script>
```

- Deep-frozen global environment (incl. frozen global object)
  - Can't update properties of Object, Array, Function, Math, JSON, etc.
- Whitelisted global environment
  - No "powerful" non-standard globals
    (e.g. document, window, XMLHttpRequest, ...)
  - Code that spawns an SES environment may provide selective access to these
- Patches `eval` and `Function` to accept only ES5/strict code, that can only name global variables on the whitelist

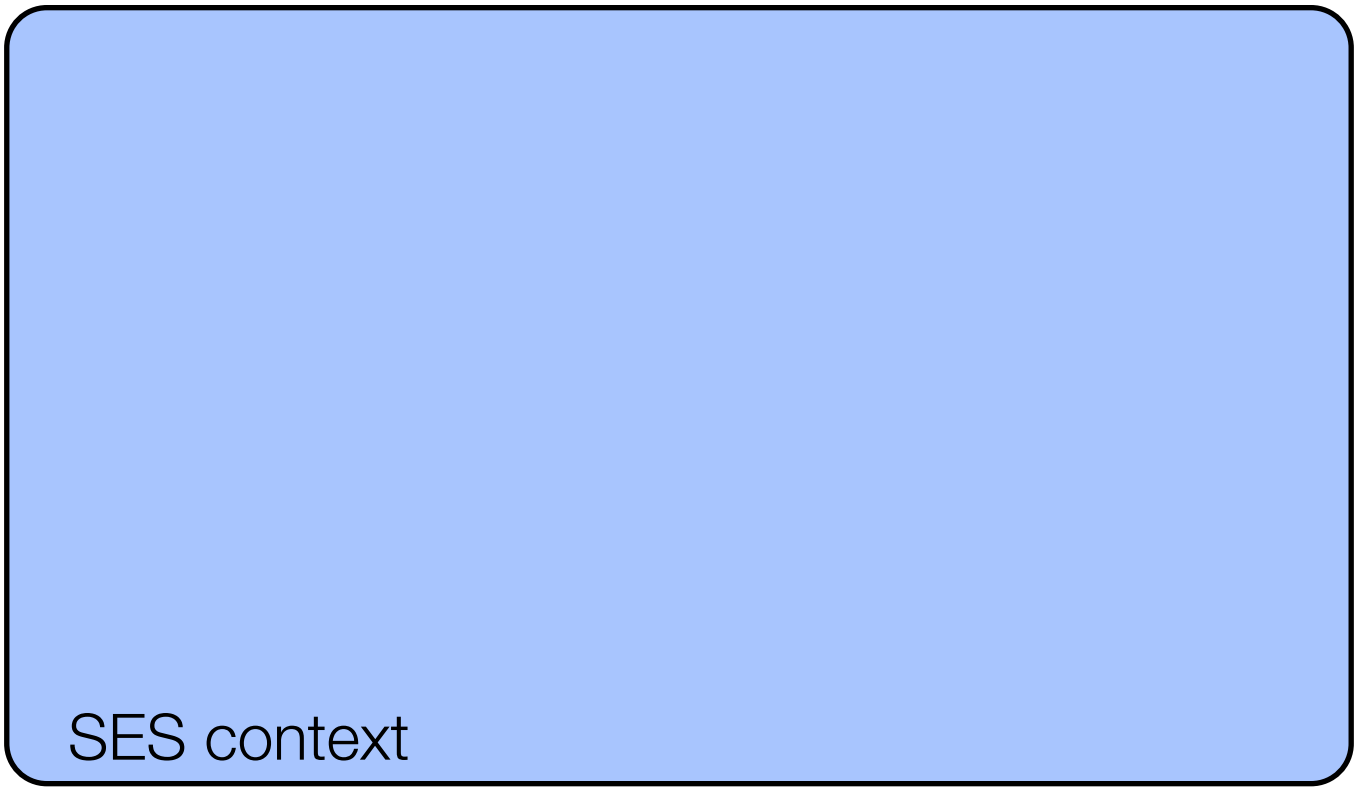
Proxies again

- 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



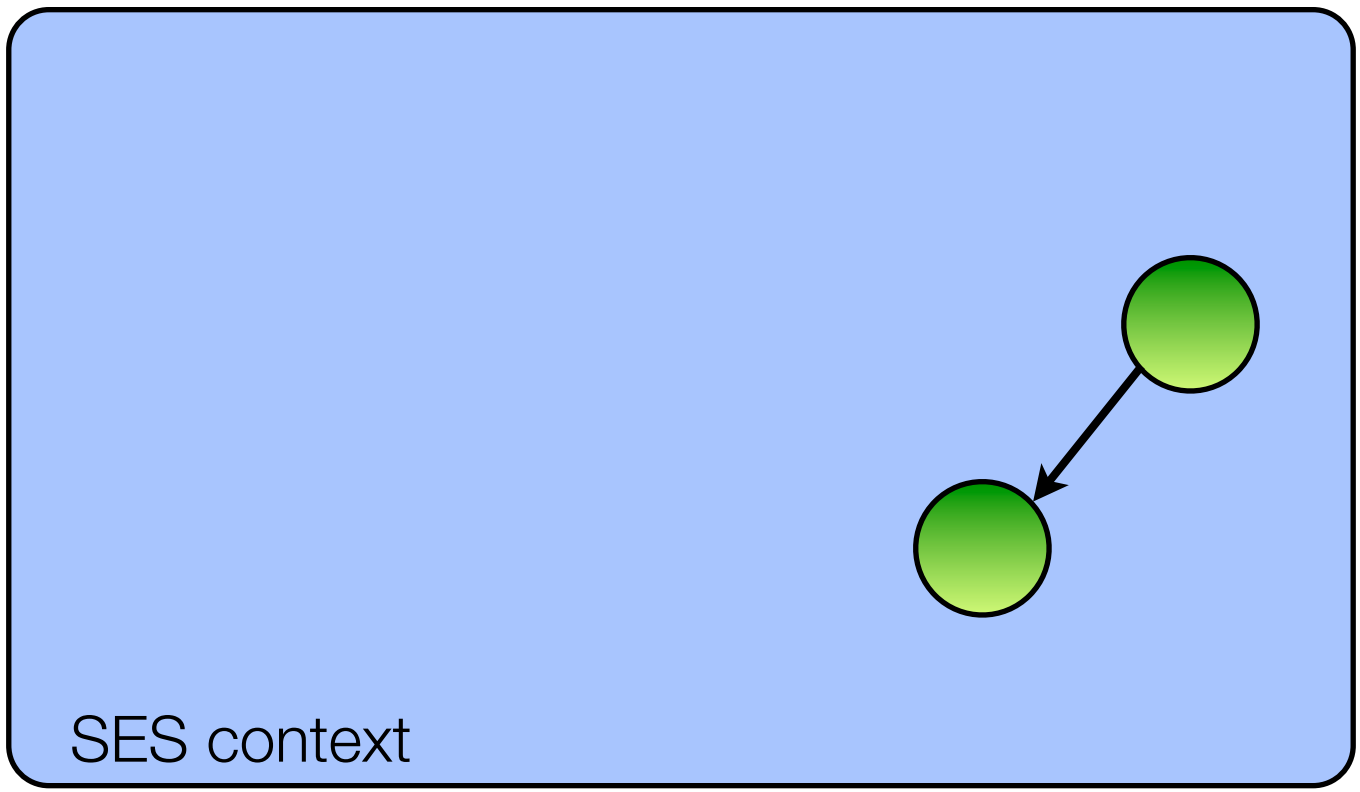
Membranes

- Transitively revocable references
- All within a single JavaScript context/frame



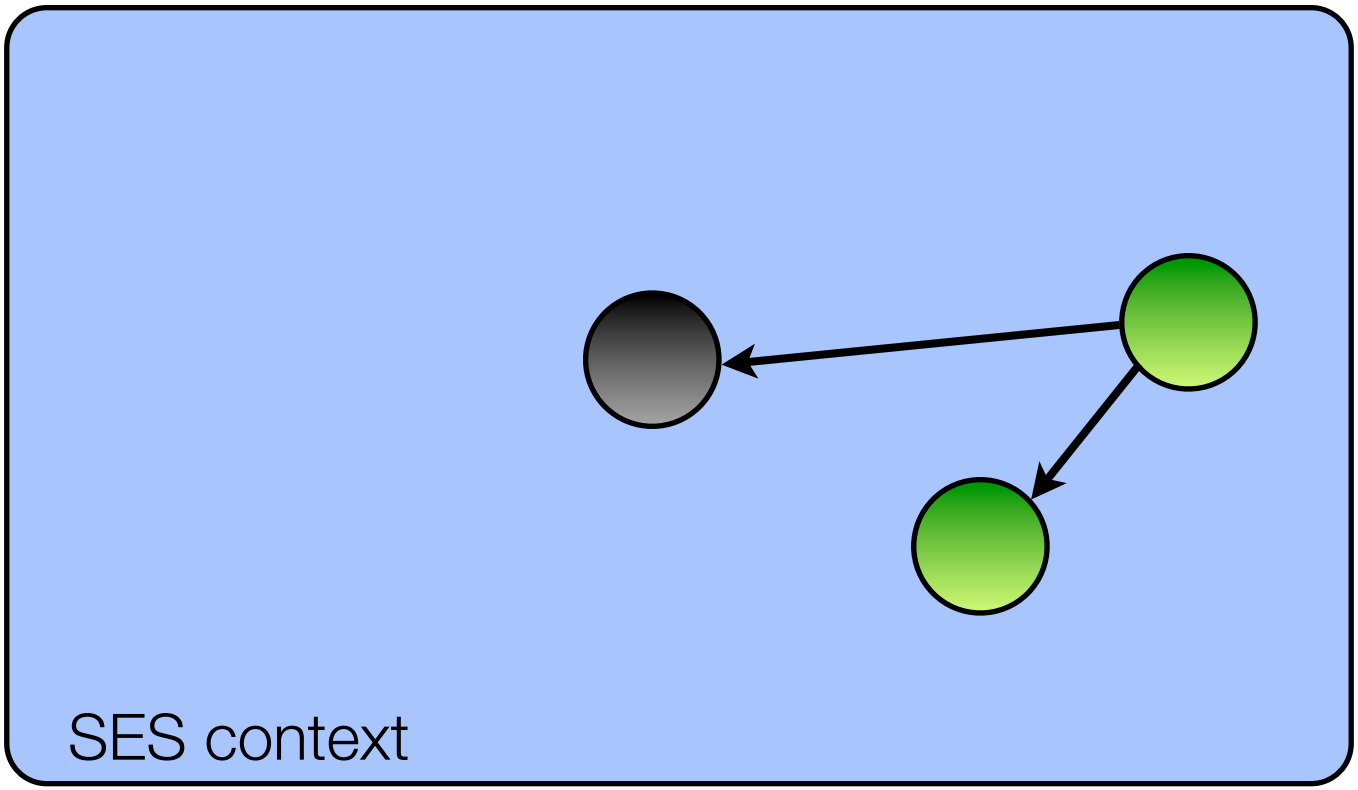
Membranes

- Transitively revocable references
- All within a single JavaScript context/frame



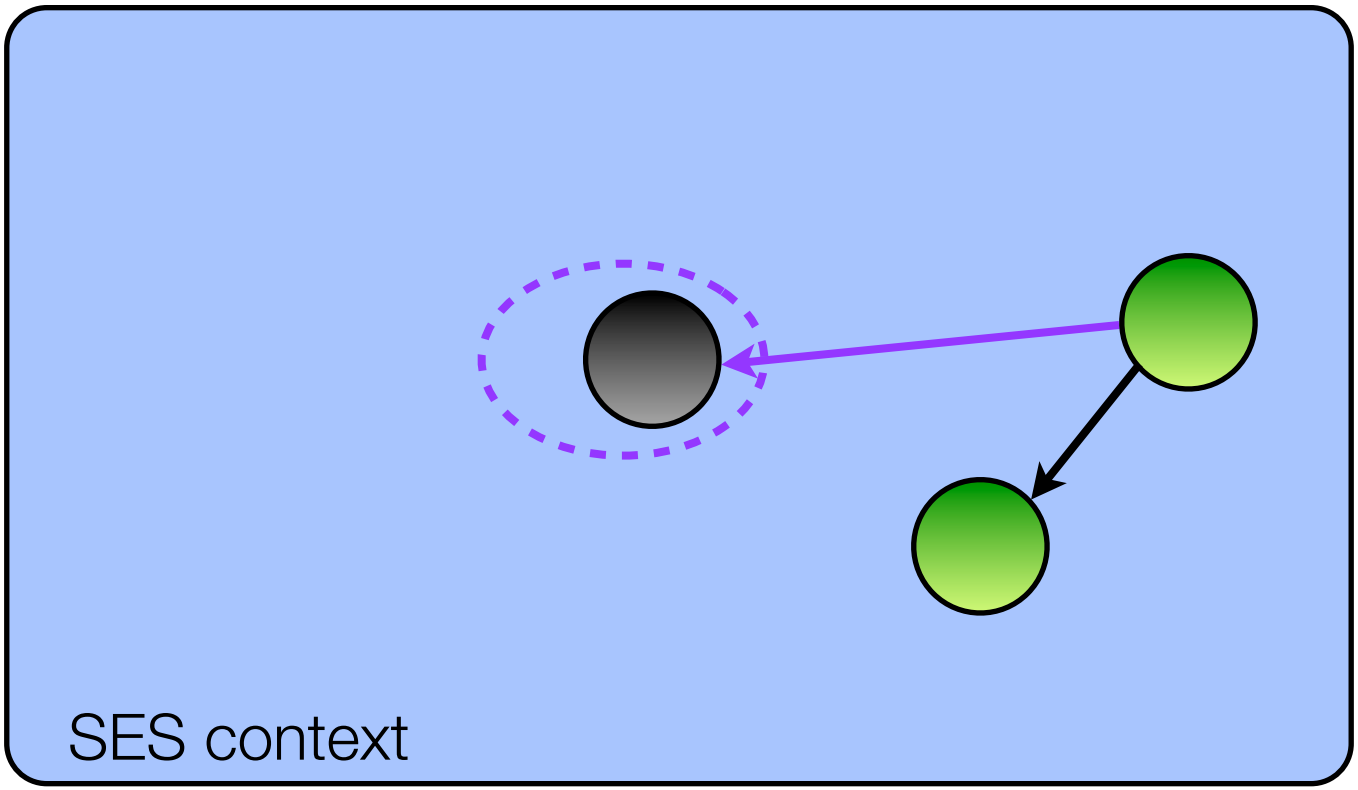
Membranes

- Transitively revocable references
- All within a single JavaScript context/frame



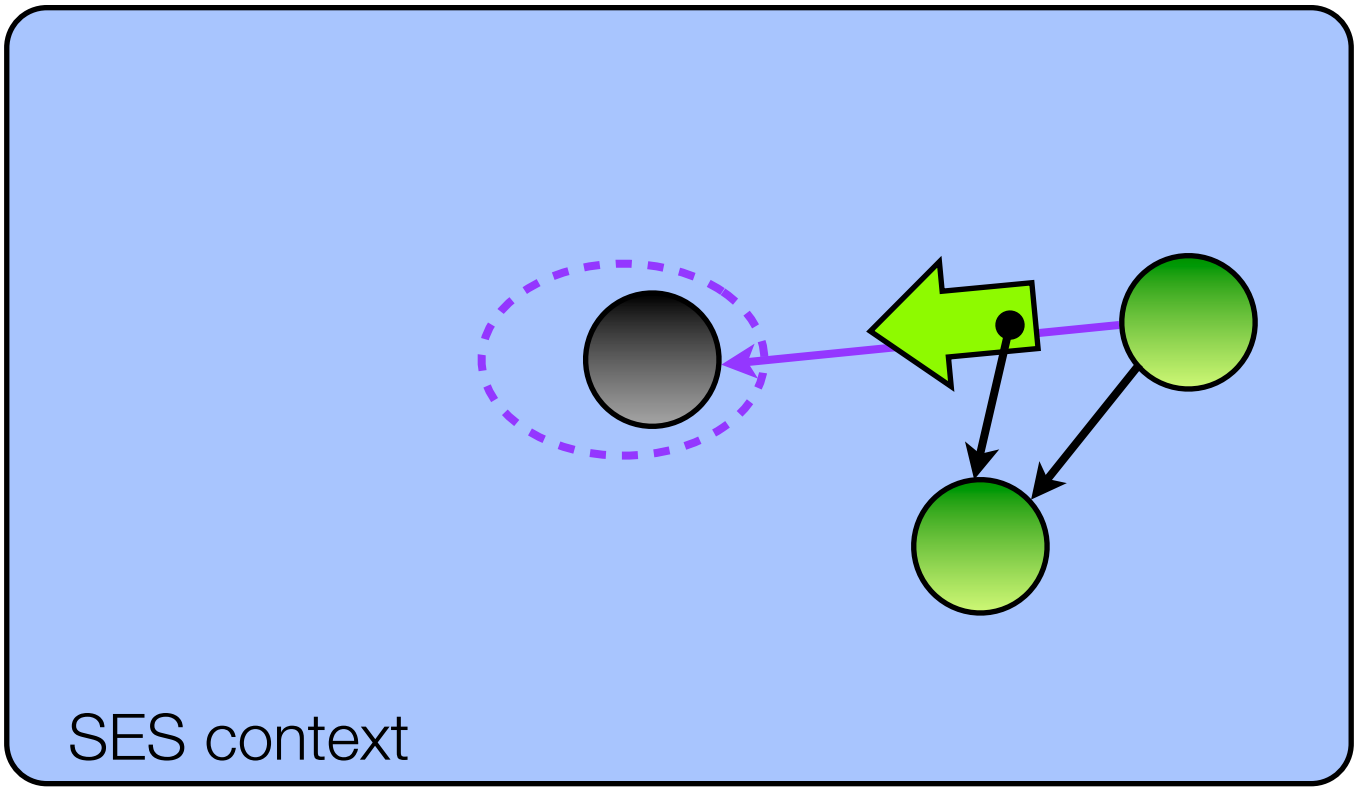
Membranes

- Transitively revocable references
- All within a single JavaScript context/frame



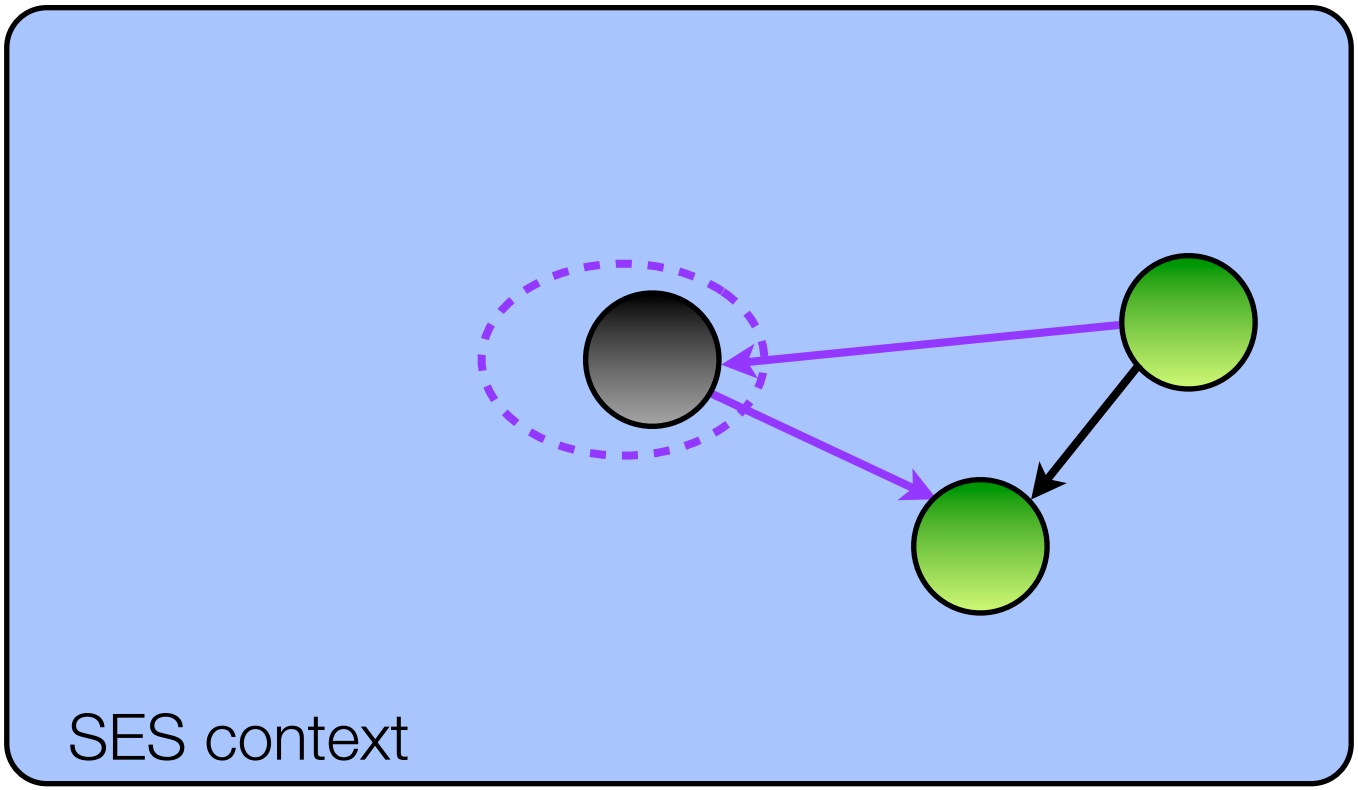
Membranes

- Transitively revocable references
- All within a single JavaScript context/frame



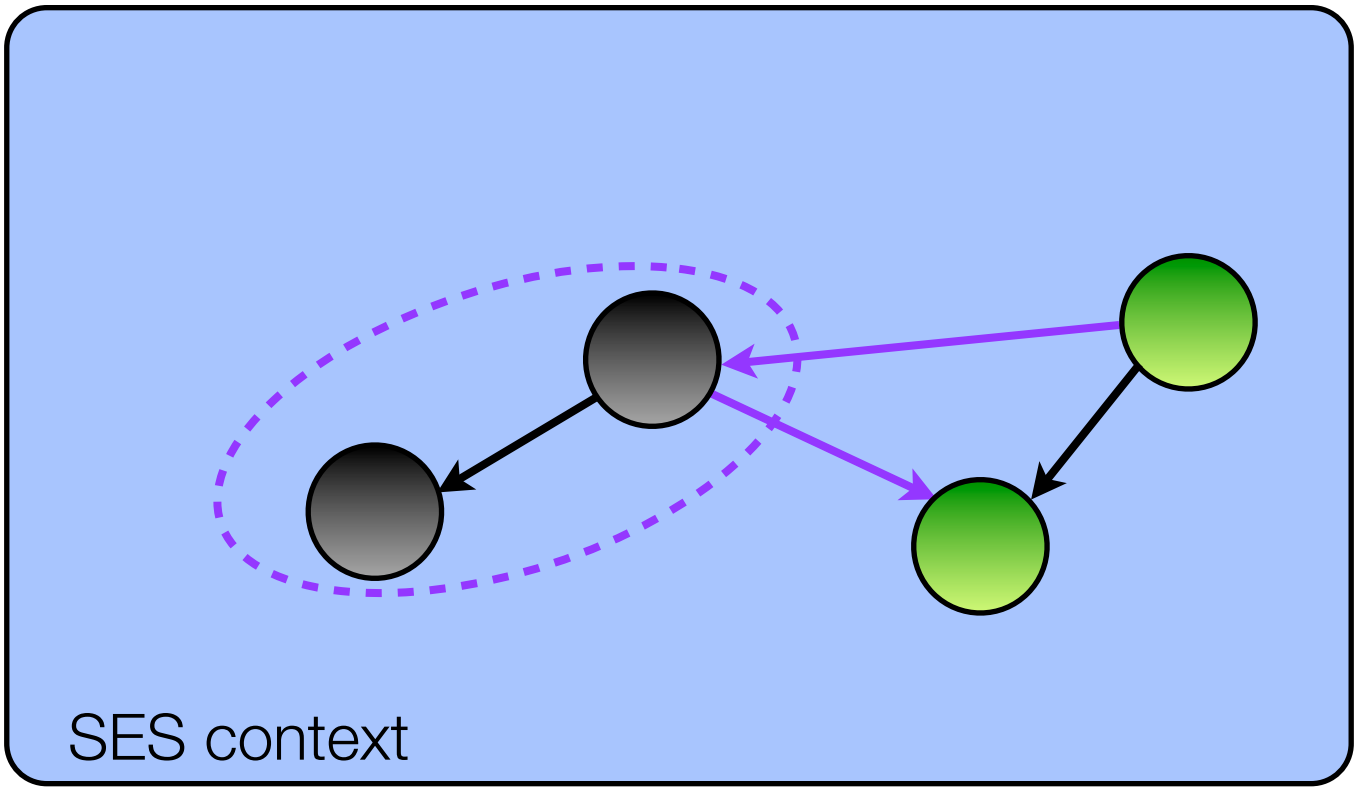
Membranes

- Transitively revocable references
- All within a single JavaScript context/frame



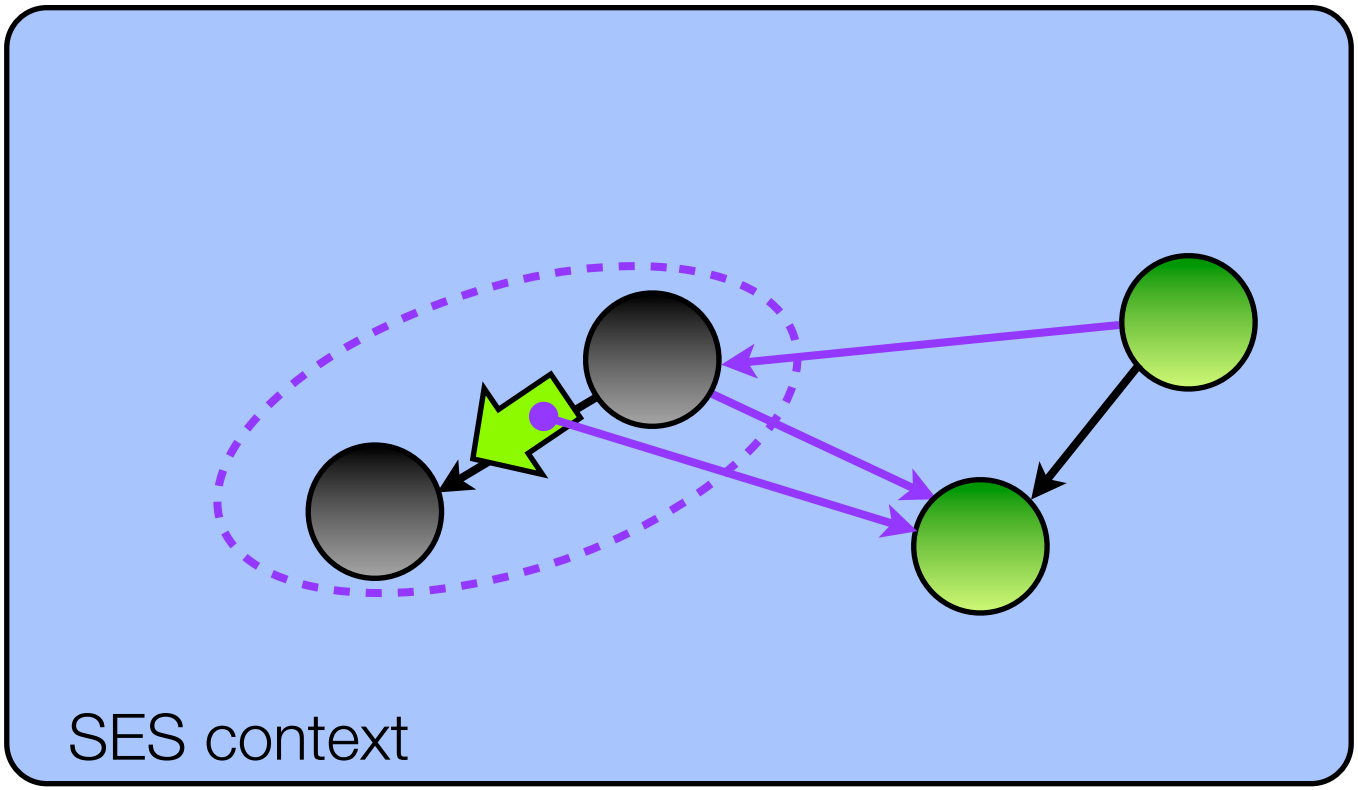
Membranes

- Transitively revocable references
- All within a single JavaScript context/frame



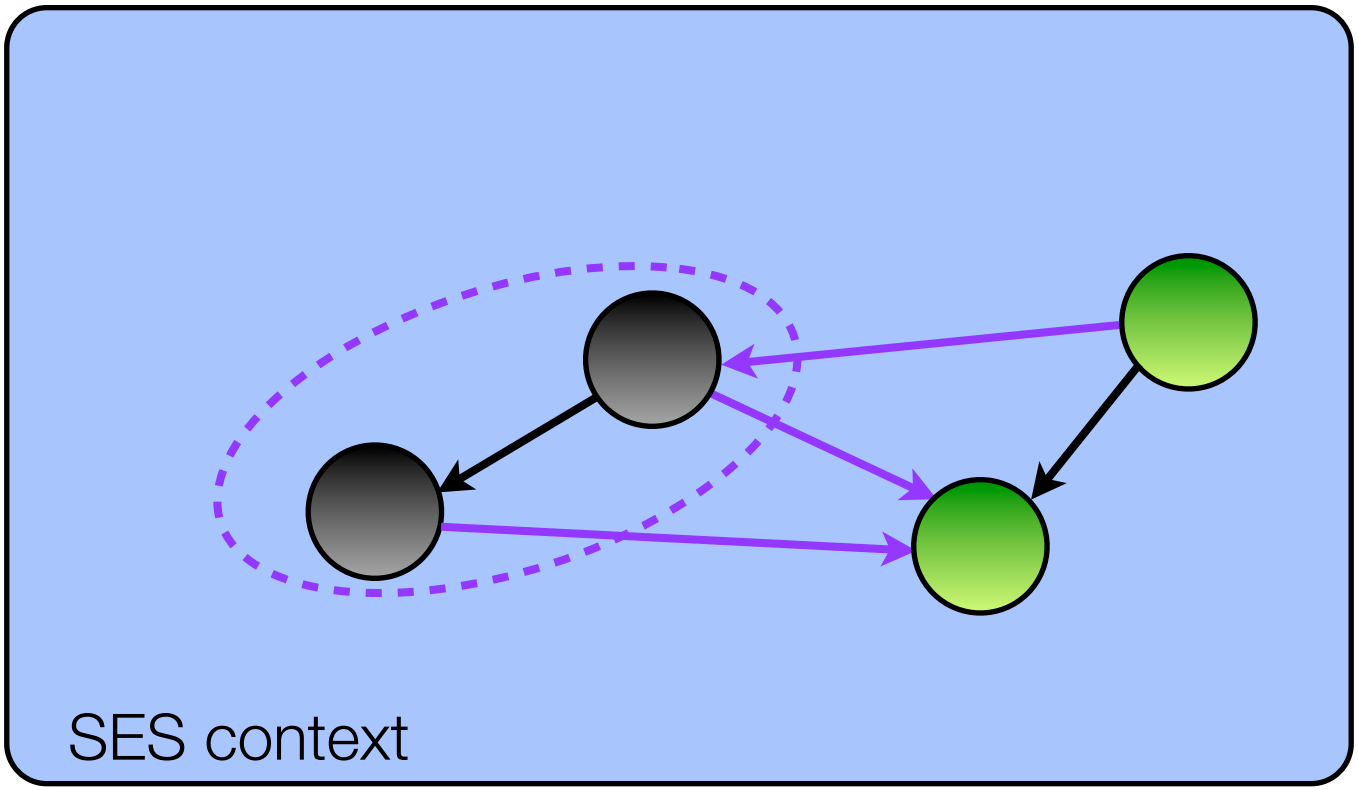
Membranes

- Transitively revocable references
- All within a single JavaScript context/frame



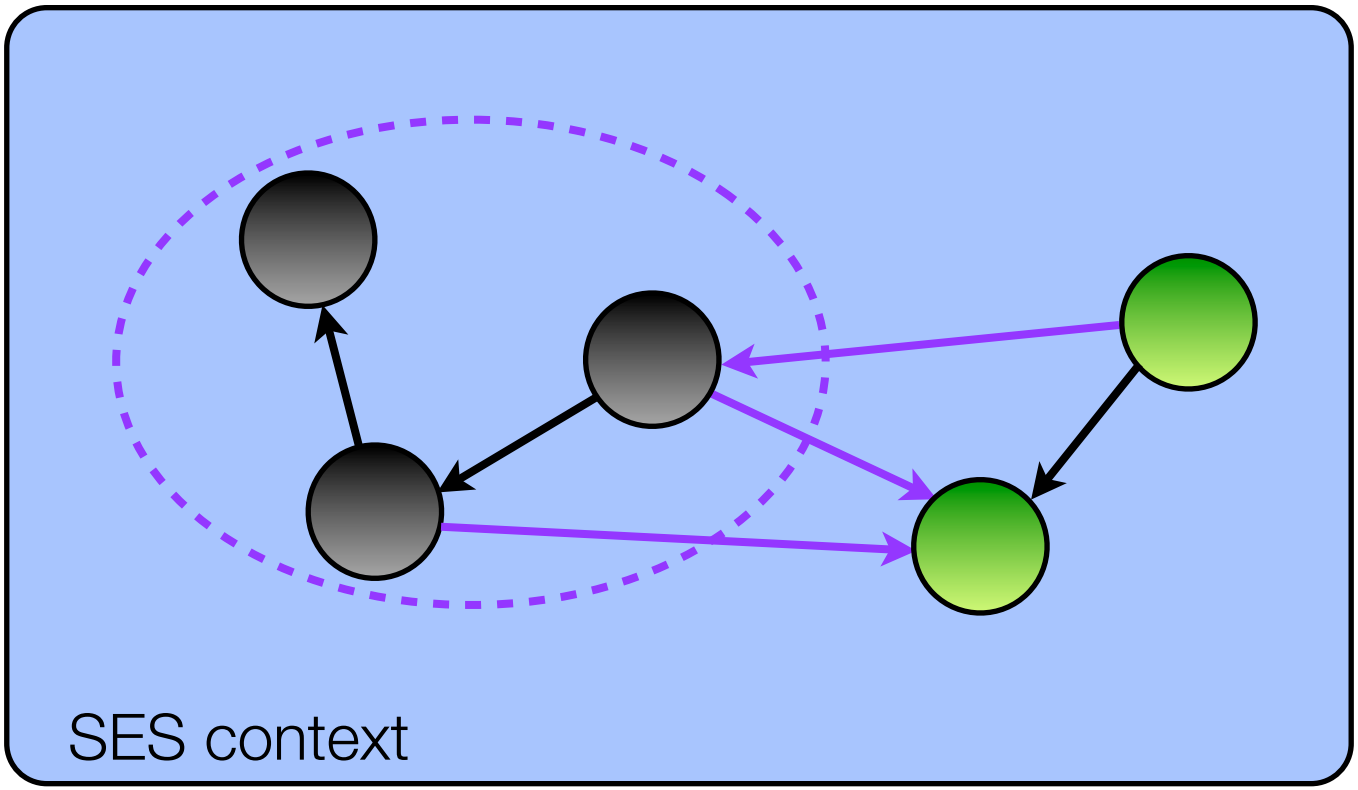
Membranes

- Transitively revocable references
- All within a single JavaScript context/frame



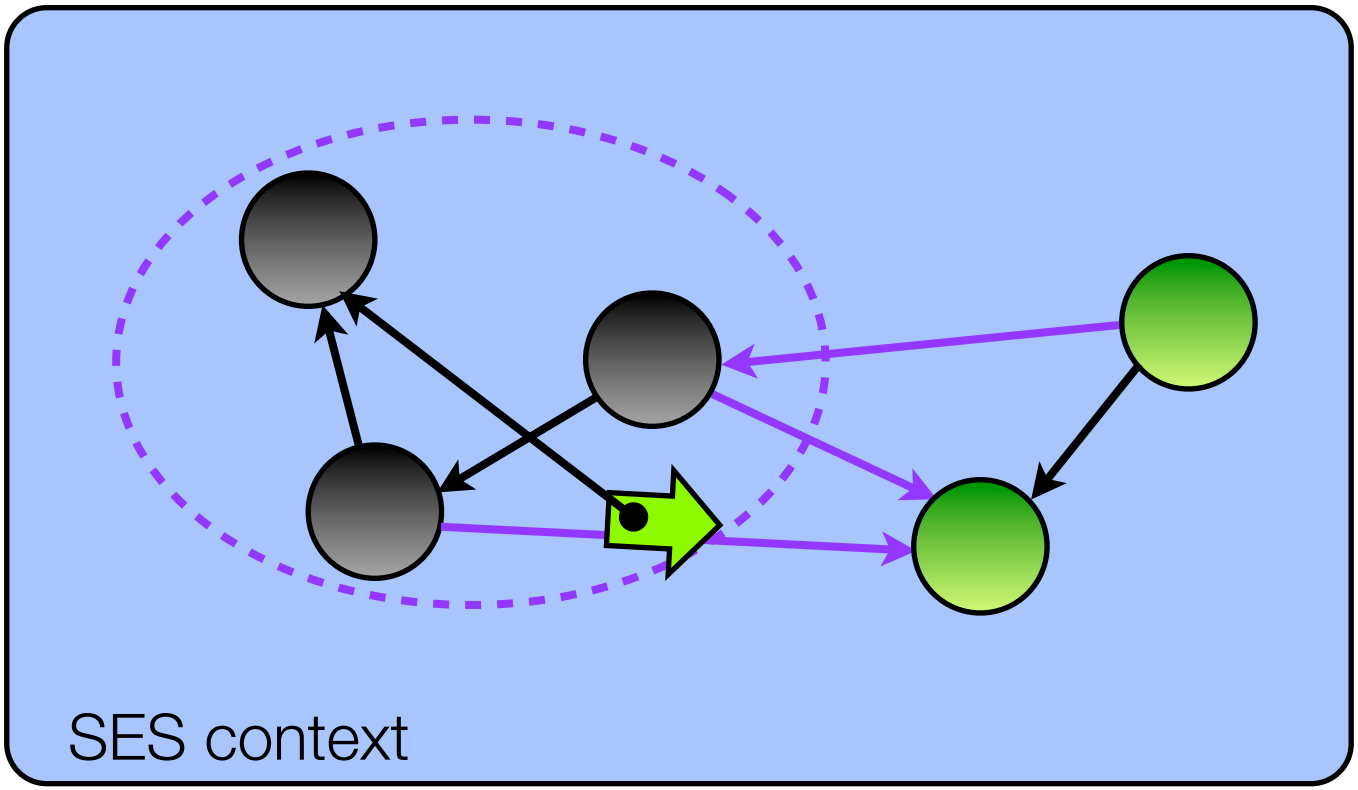
Membranes

- Transitively revocable references
- All within a single JavaScript context/frame



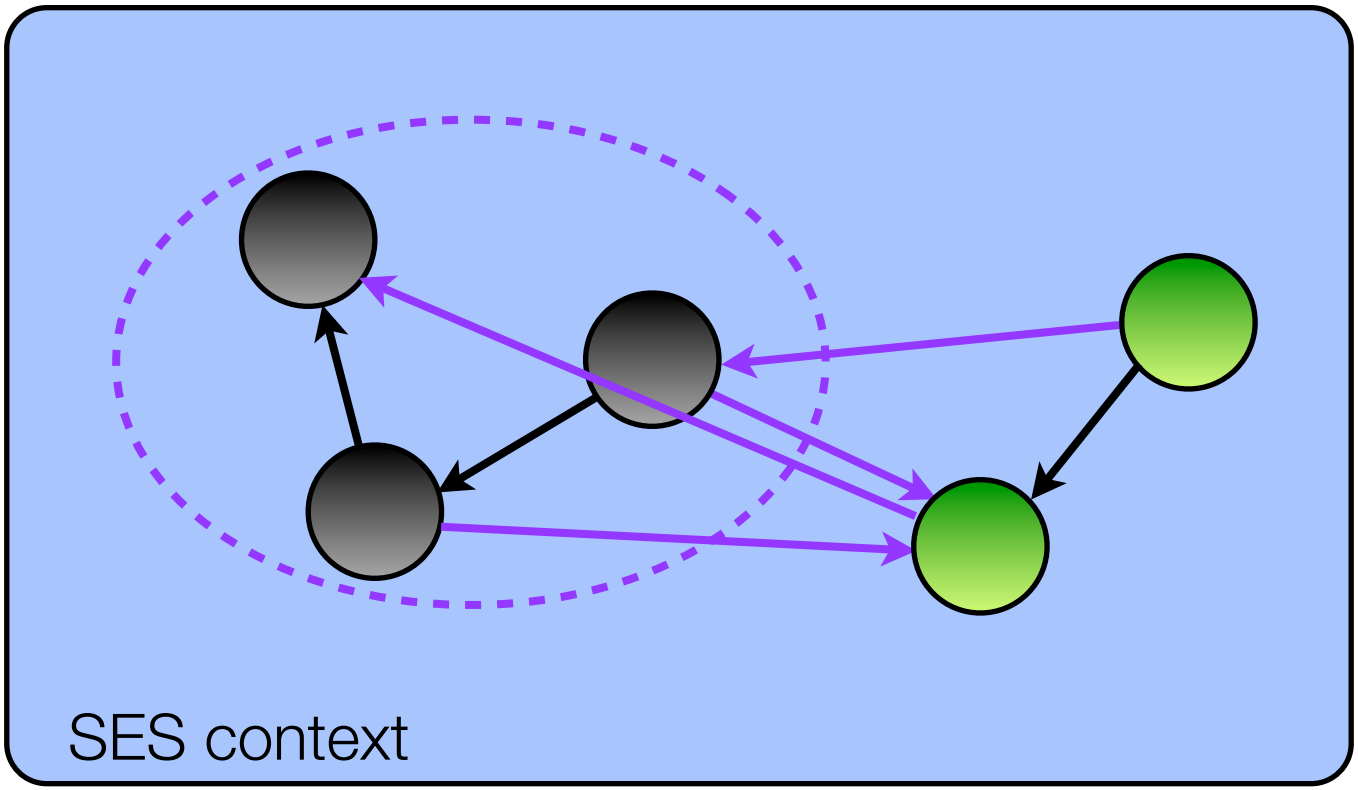
Membranes

- Transitively revocable references
- All within a single JavaScript context/frame



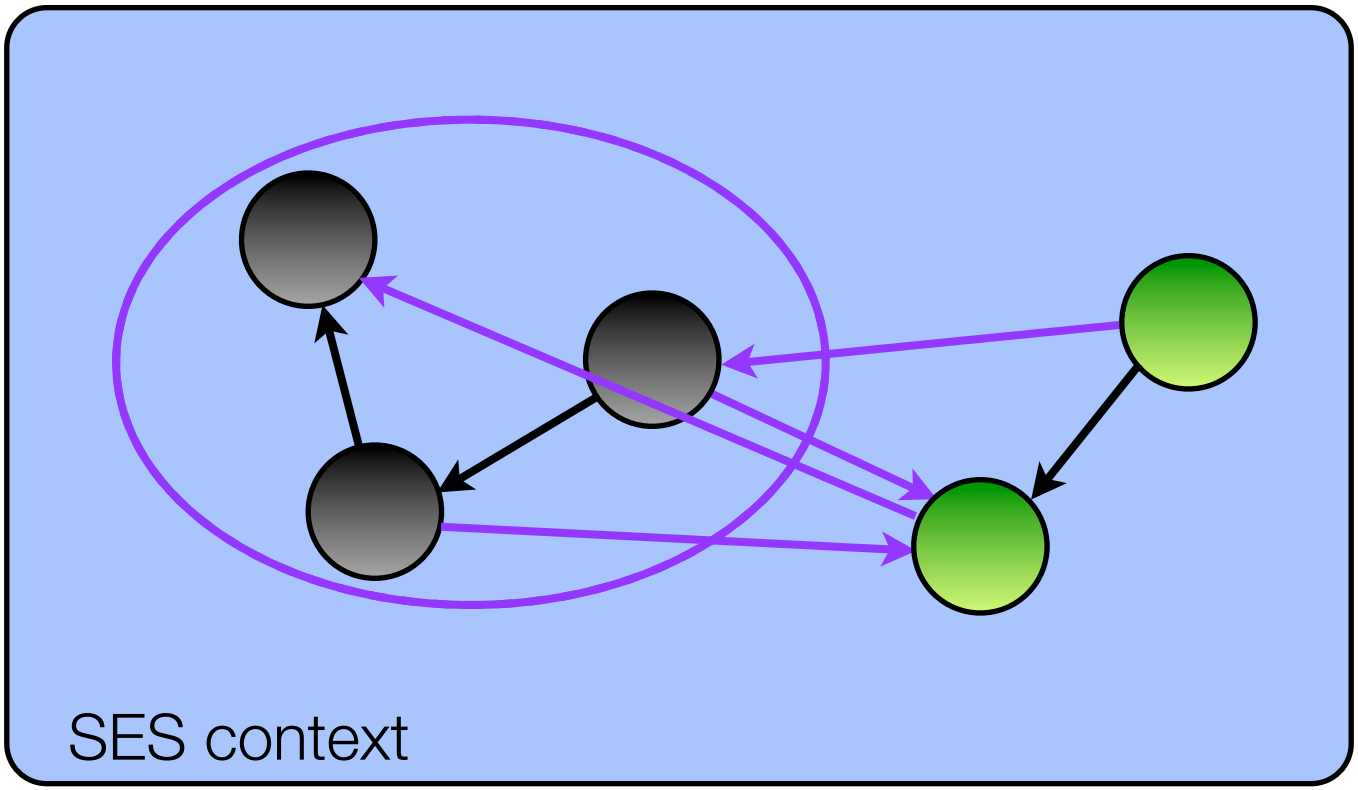
Membranes

- Transitively revocable references
- All within a single JavaScript context/frame



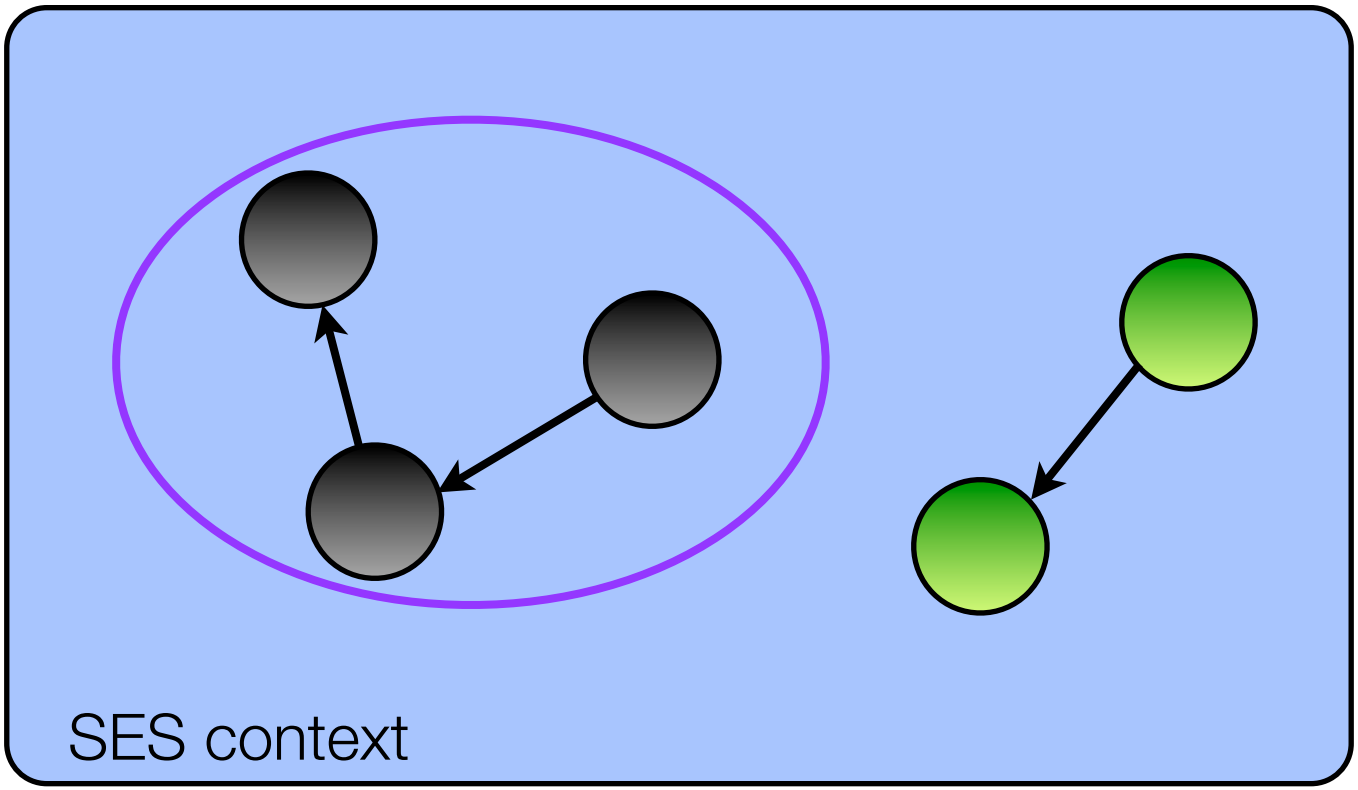
Membranes

- Transitively revocable references
- All within a single JavaScript context/frame



Membranes

- Transitively revocable references
- All within a single JavaScript context/frame



Wrap-up

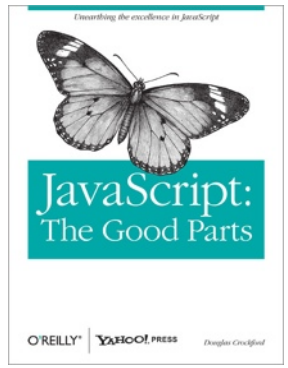
Wrap-up

ES3

ES5

ES5/strict

SES



JavaScript:

the Good,
the Bad,

the Strict,

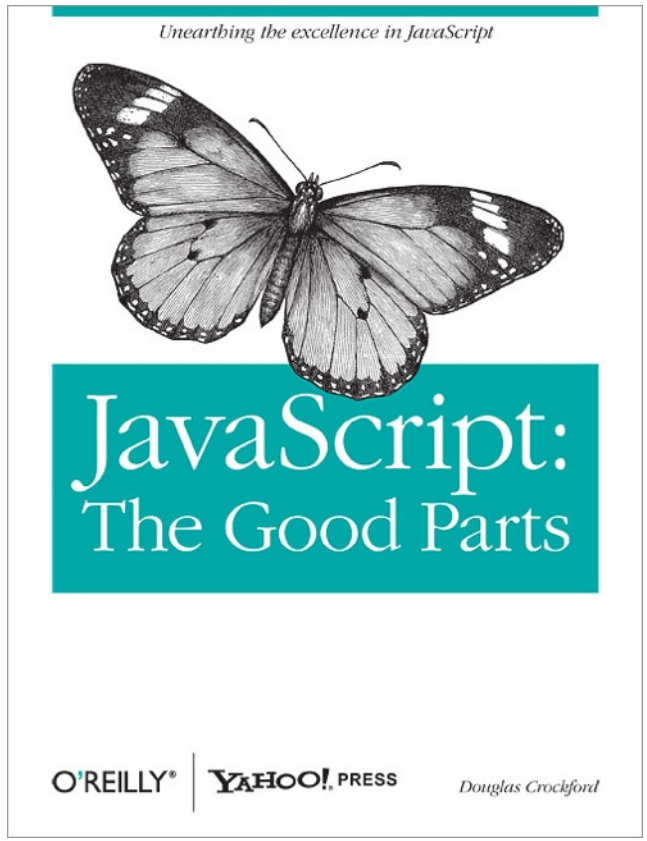
and
the Secure parts.

Take-home messages

- Secure ECMAScript (SES) builds on ES5 strict mode
- If you want your code to be *securable*, opt into strict mode
- Proxies are a power-tool to express fine-grained security policies (e.g. membranes)

References

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



References

- ECMAScript 5:
  - “Changes to JavaScript Part 1: EcmaScript 5” (Mark S. Miller, Waldemar Horwat, Mike Samuel), Google Tech Talk (May 2009)
  - “Secure Mashups in ECMAScript 5” (Mark S. Miller), QCon Talk http://www.infoq.com/presentations/Secure-Mashups-in-ECMAScript-5
- Caja: https://developers.google.com/caja
- SES: http://code.google.com/p/google-caja/wiki/SES
- Proxies: http://soft.vub.ac.be/~tvcutsem/invokedynamic/proxies_tutorial
- ES6 latest developments: http://wiki.ecmascript.org and the es-discuss@mozilla.org mailing list