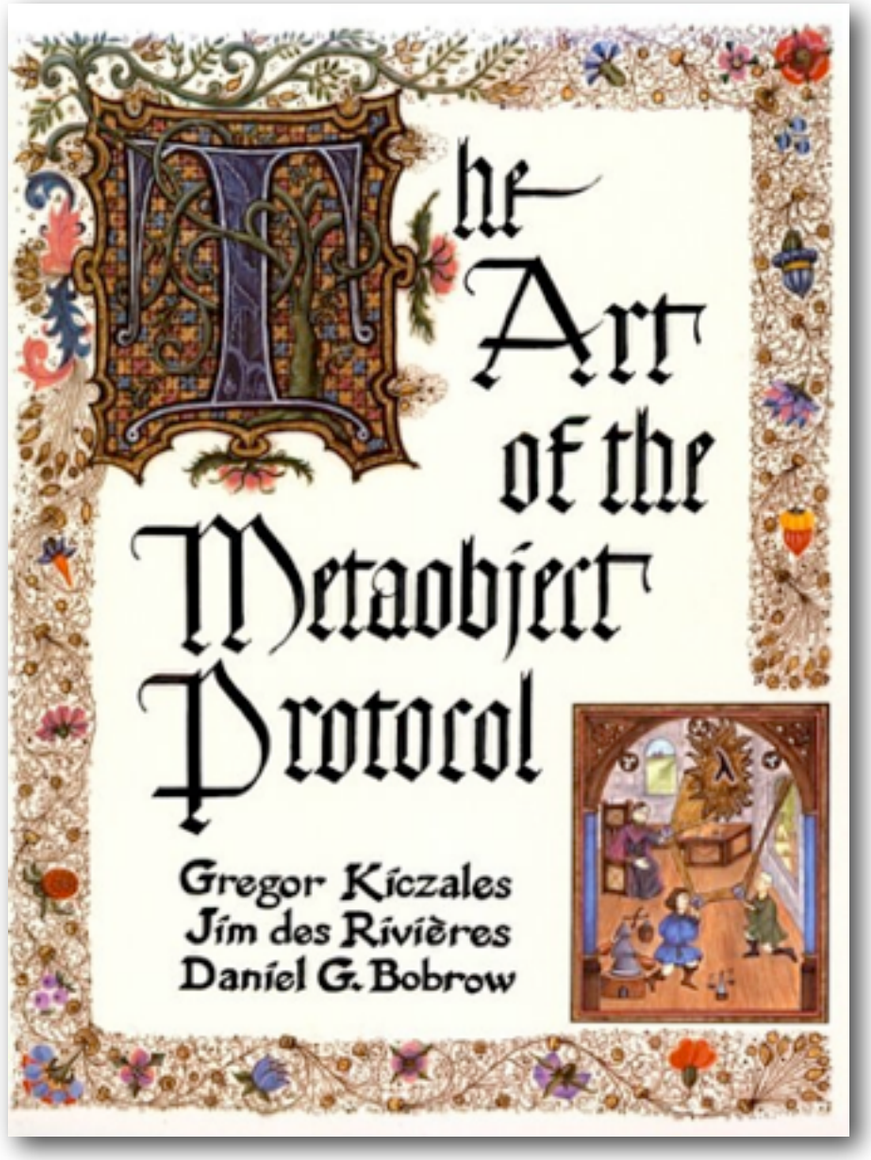
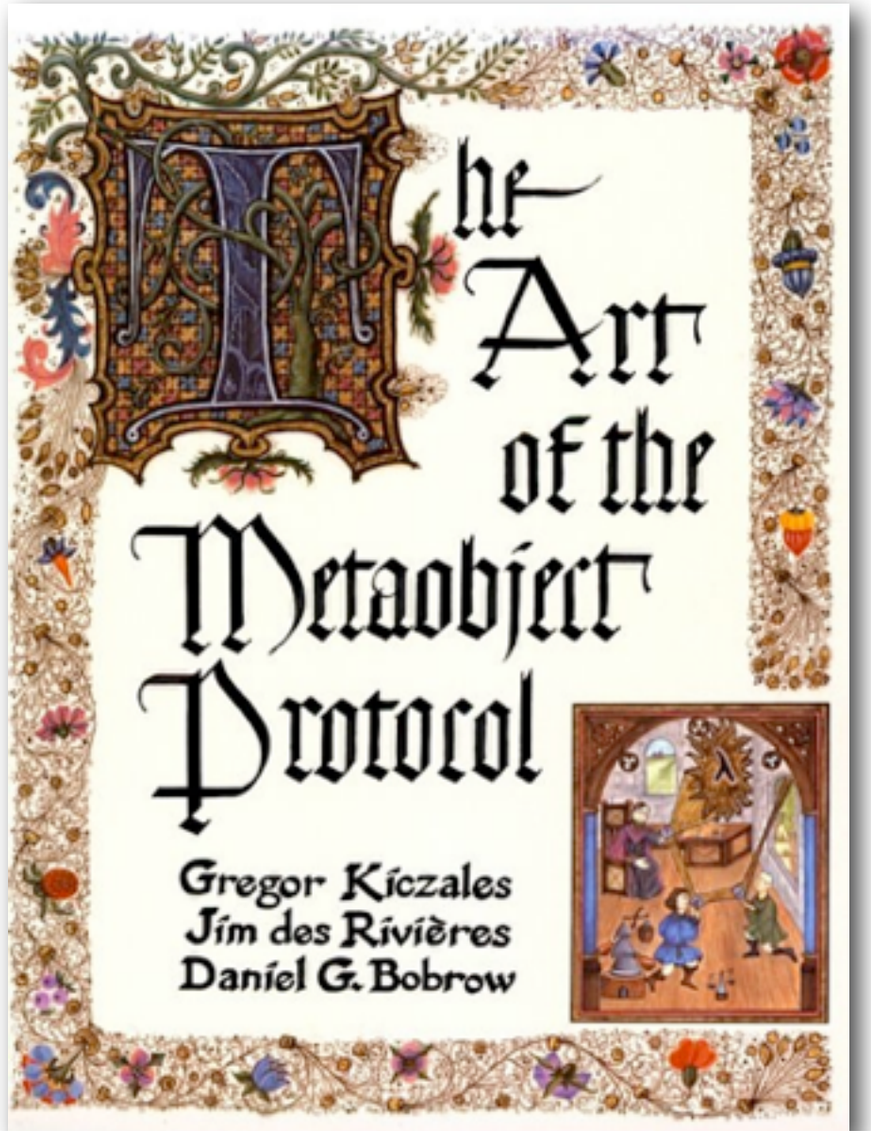
Tradeoffs in language design: The case of Javascript proxies

Tom Van Cutsem
(joint work with Mark S. Miller, with feedback from many others)

What do these have in common?



What do these have in common?

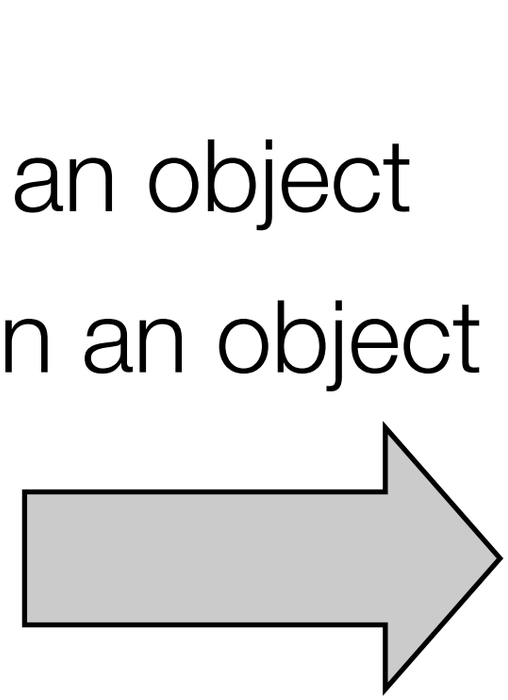


VIRTUALIZATION

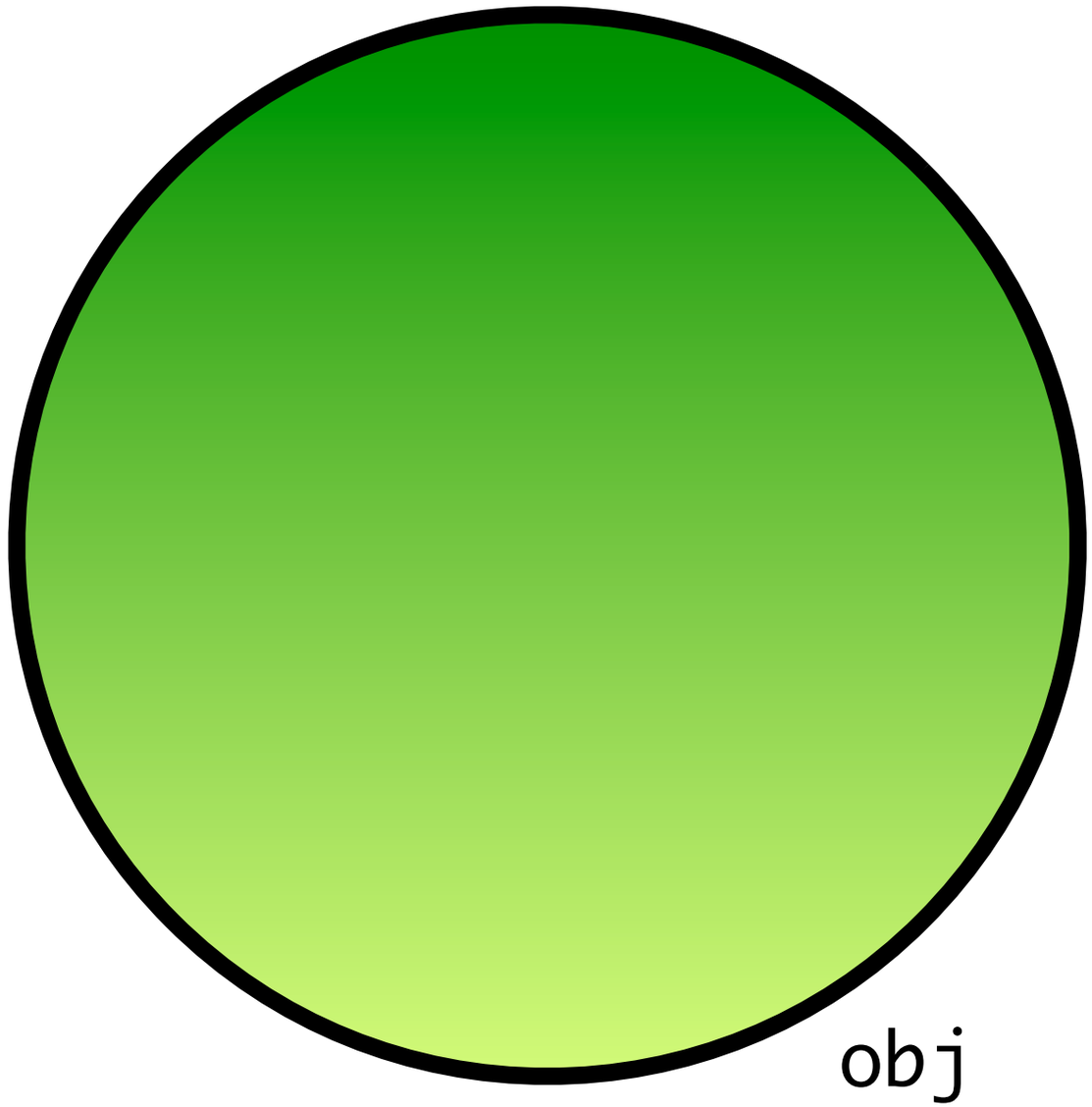
Virtualizing objects

querying an object

acting upon an object

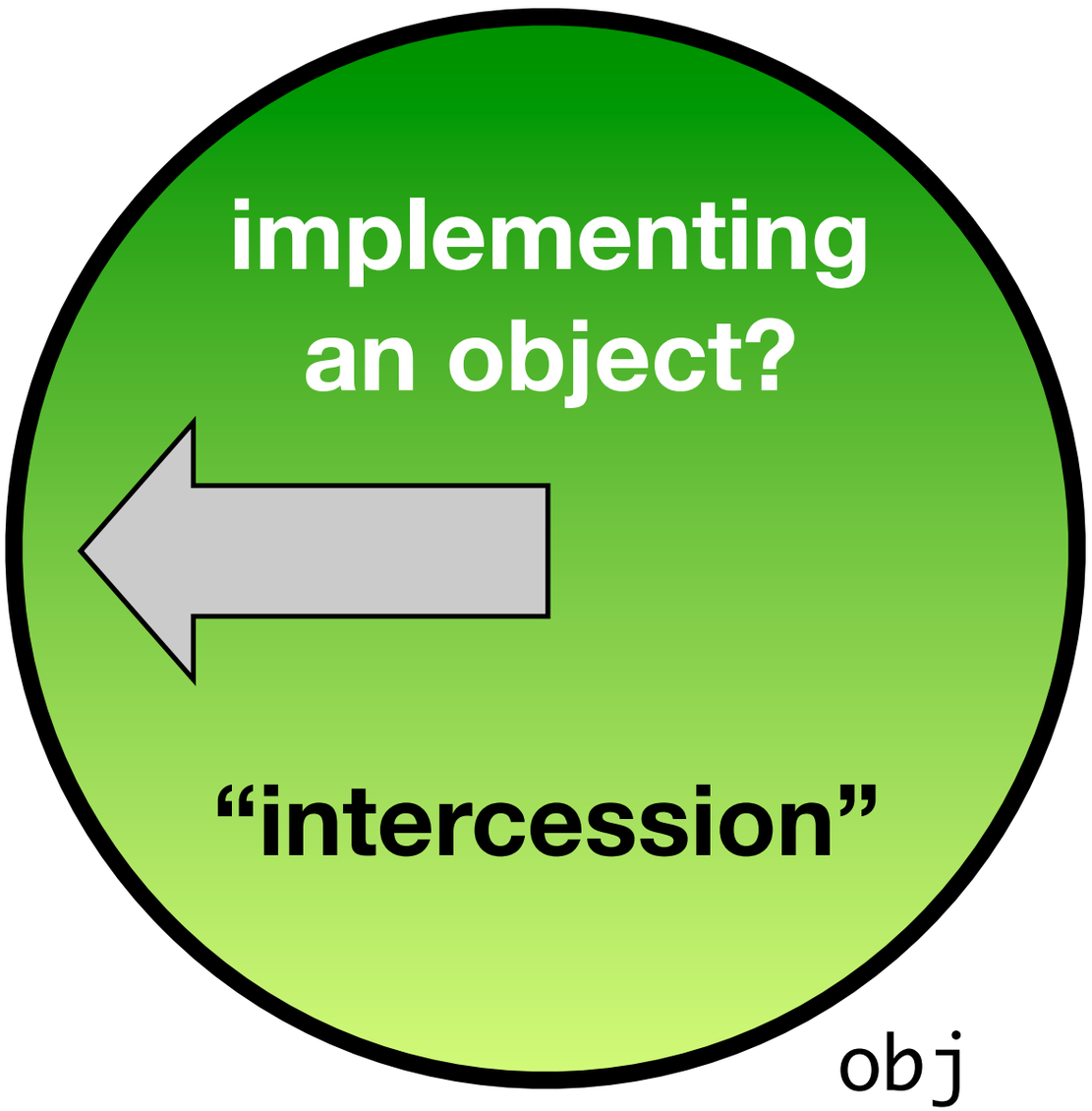


“introspection”




```
obj[“x”]
delete obj.x
“x” in obj
```

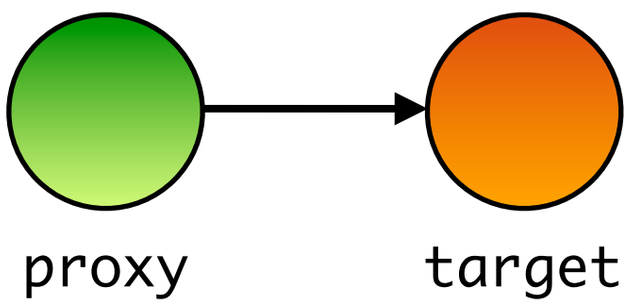
Virtualizing objects



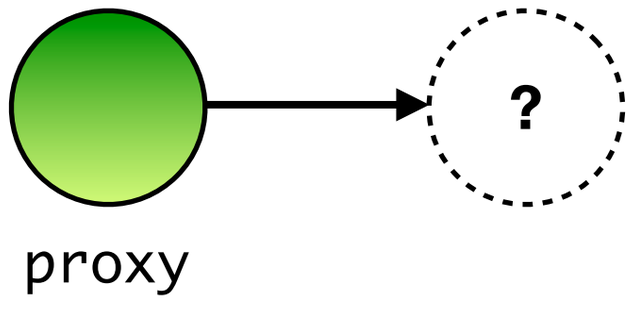

```
obj[“x”]
delete obj.x
“x” in obj
```

Why implement your own objects?

- Generic wrappers around existing objects: access control wrappers (security), tracing, profiling, contracts, taint tracking, decorators, adaptors, ...

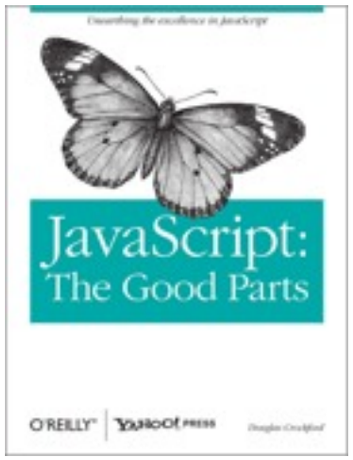


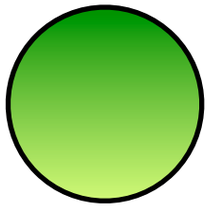
- Virtual objects: remote objects, mock objects, persistent objects, promises / futures, lazily initialized objects, ...



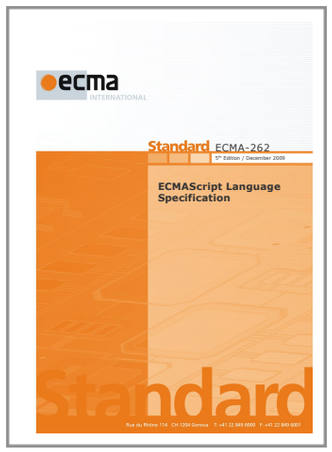
The Javascript object zoo

Native objects
(provided by ECMAScript engine)





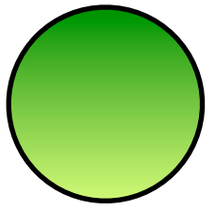
an Object



an Array

W3C®

The “DOM”

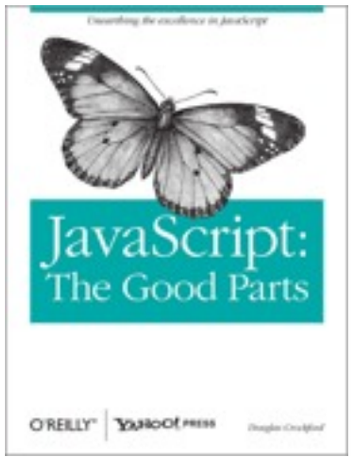


a NodeList

Normal objects
(implementable in Javascript)

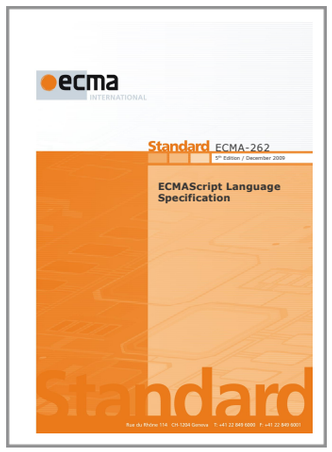
Host objects
(provided by the embedding environment, usually the browser)

The Javascript object zoo



+ proxies

an Object



an Array

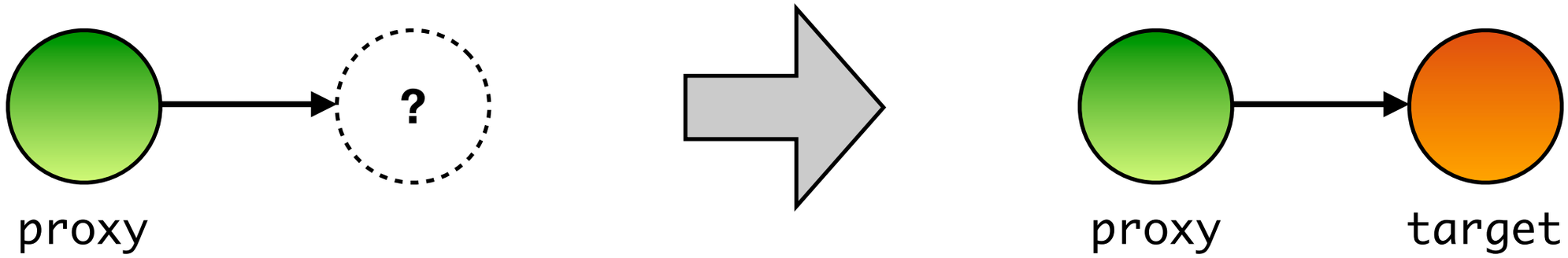
W3C®

The “DOM”

a NodeList

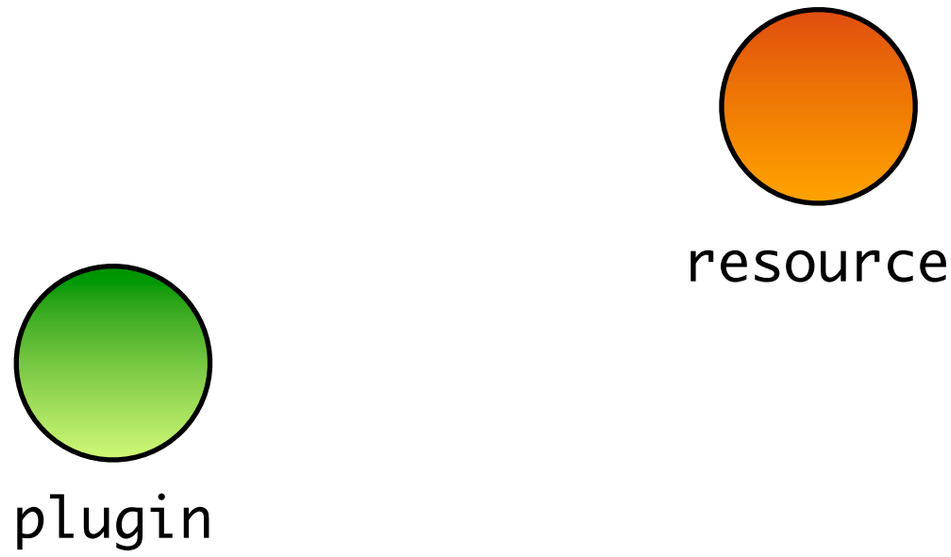
can be implemented using proxies

The rest of this talk



Example: revocable references

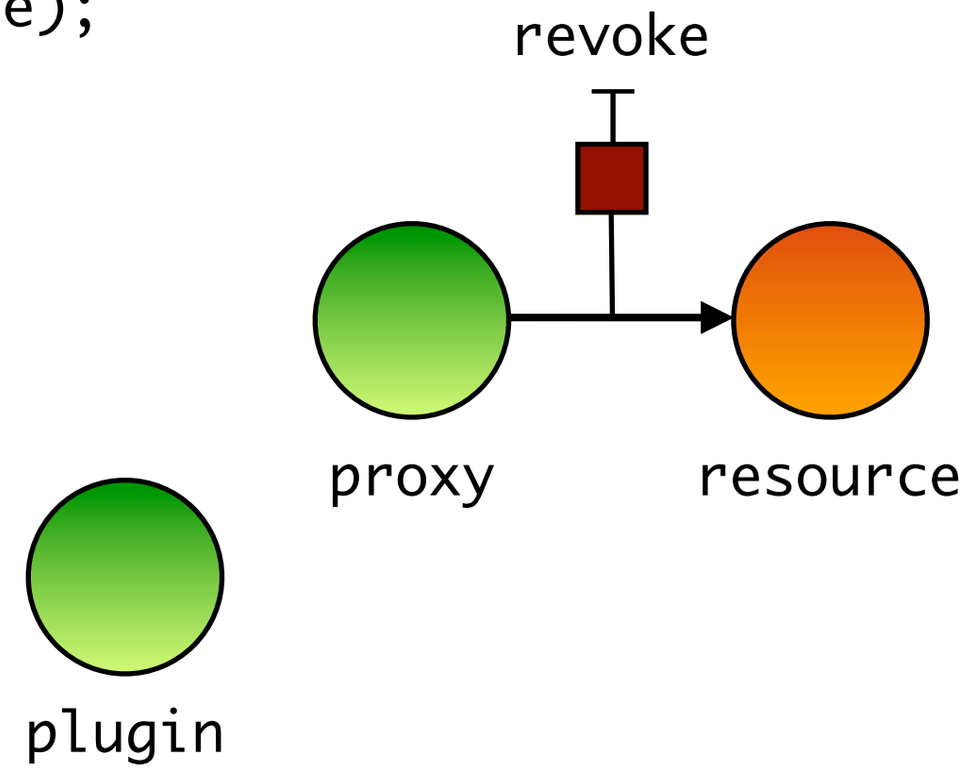
- Provide temporary access to a resource
- Useful for explicit memory management or expressing security policy



Example: revocable references

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

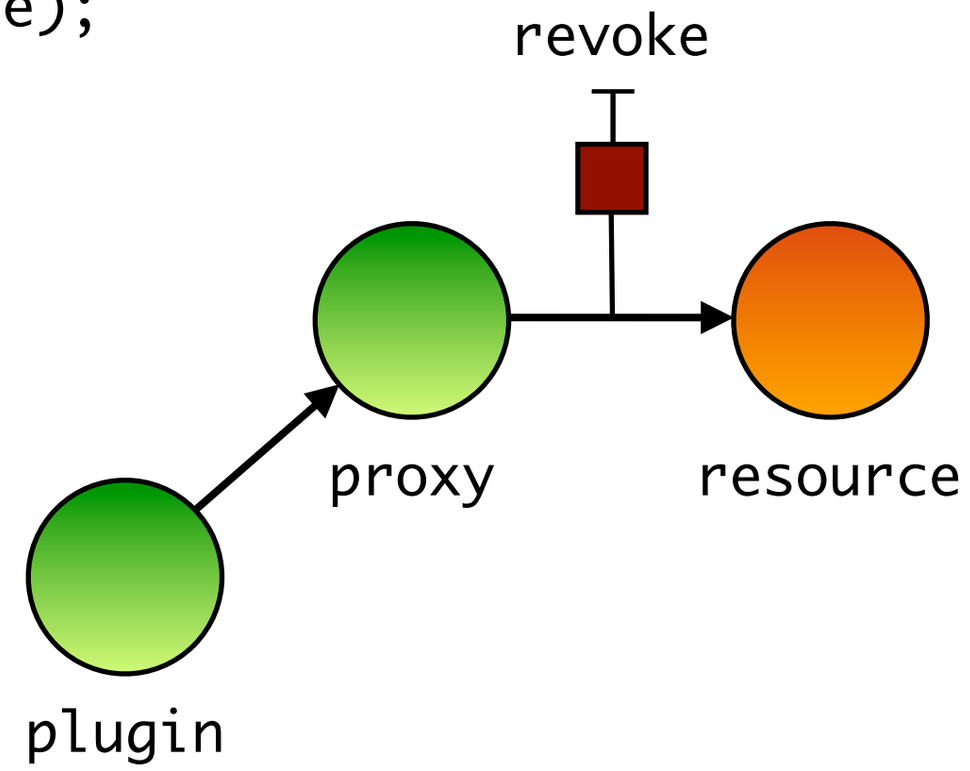
```
var {proxy,revoke} = makeRevocable(resource);
```



Example: revocable references

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

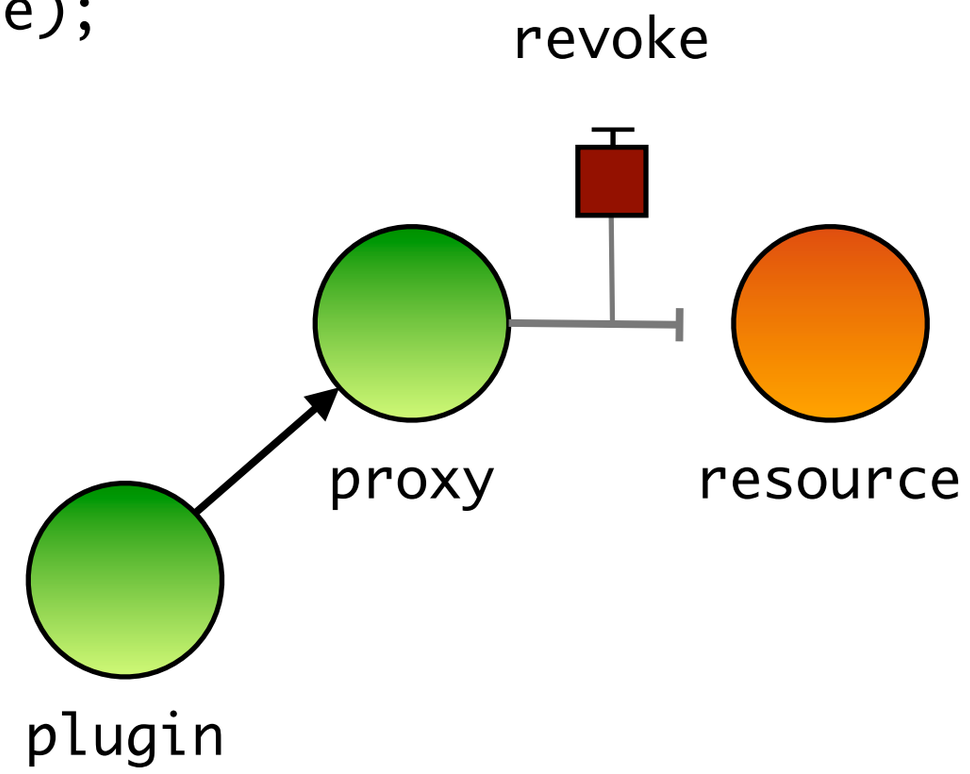
```
var {proxy,revoke} = makeRevocable(resource);

plugin.give(proxy)
```



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

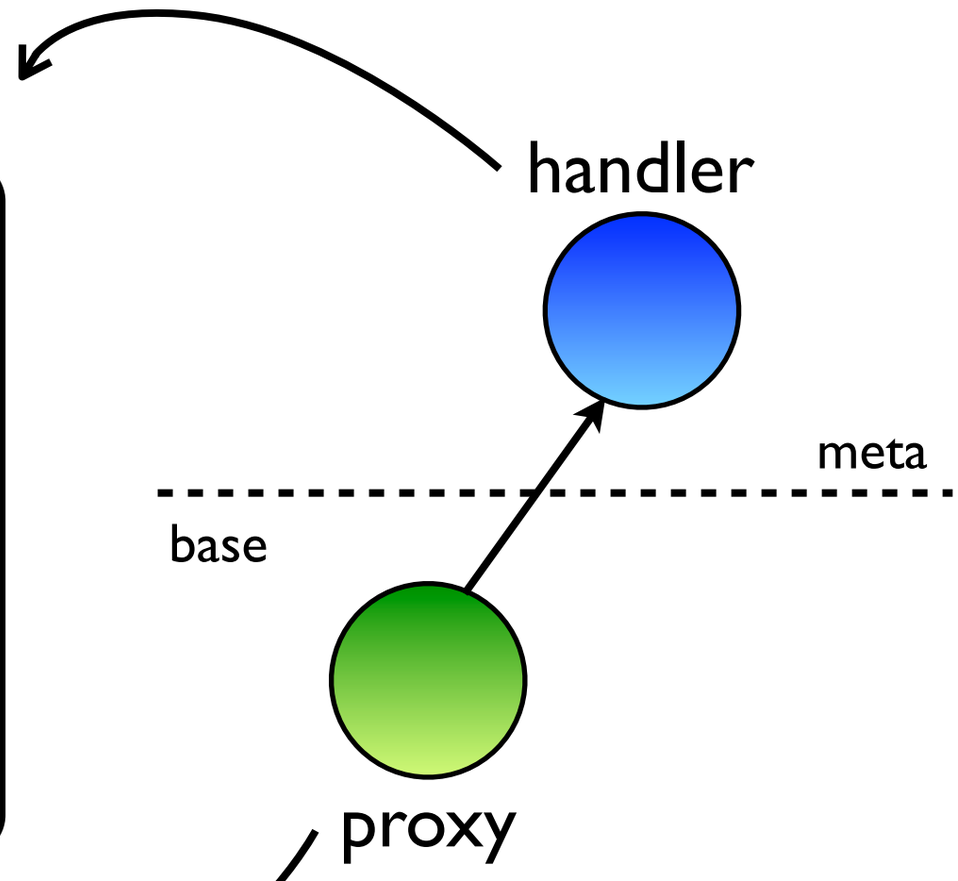
```
function makeRevocable(target) {
  var enabled = true;
  var proxy = Proxy({
  });
  return {
     proxy: proxy,
    revoke: function() { enabled = false; }
  }
}
```

Revocable references

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

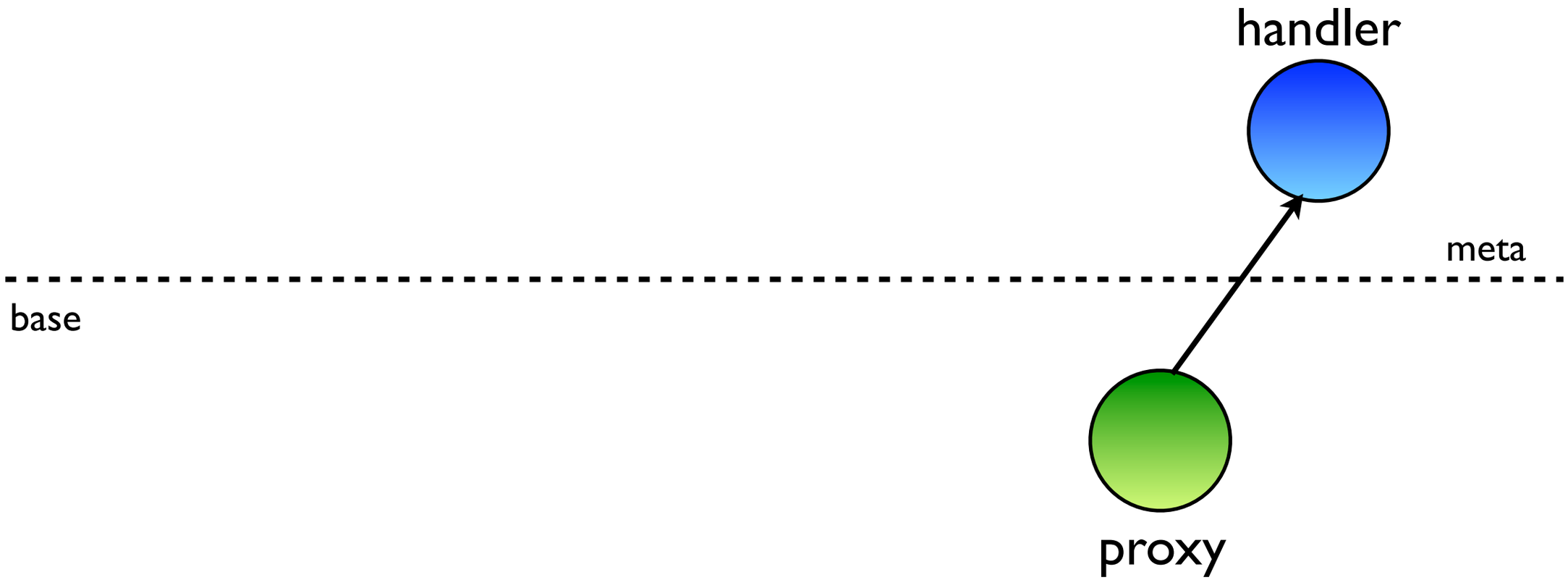
Revocable references

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



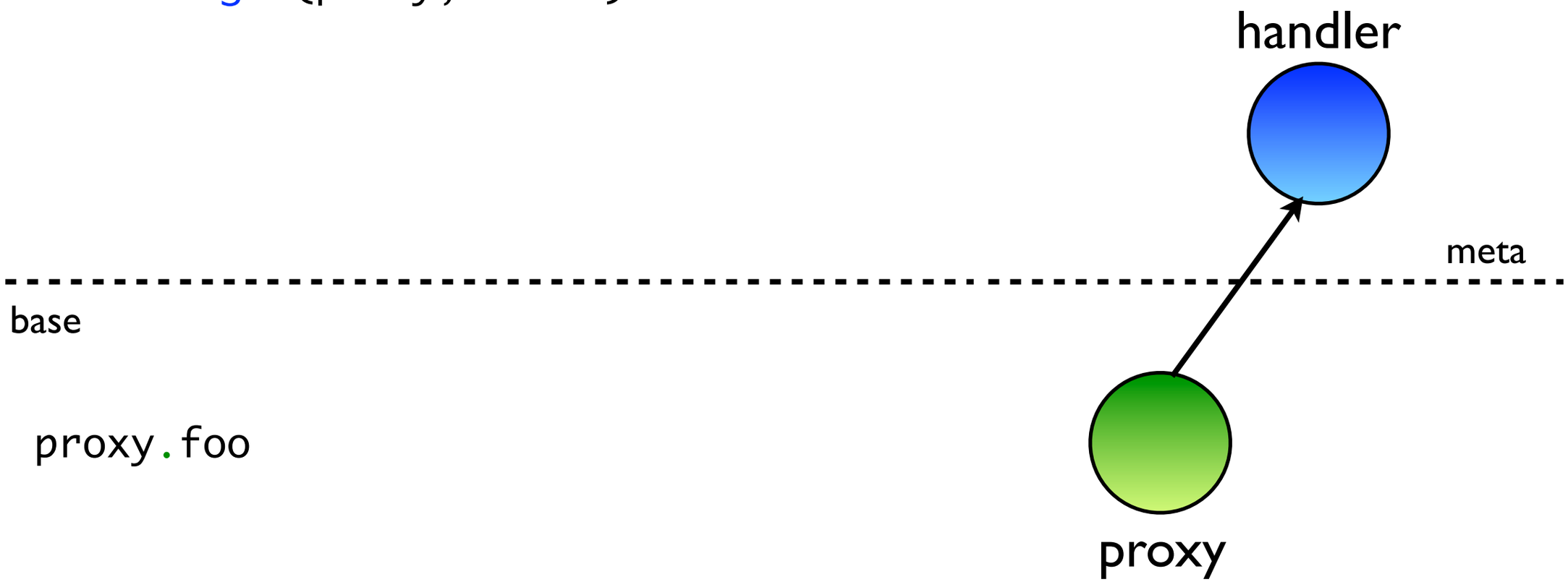
Stratified API

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



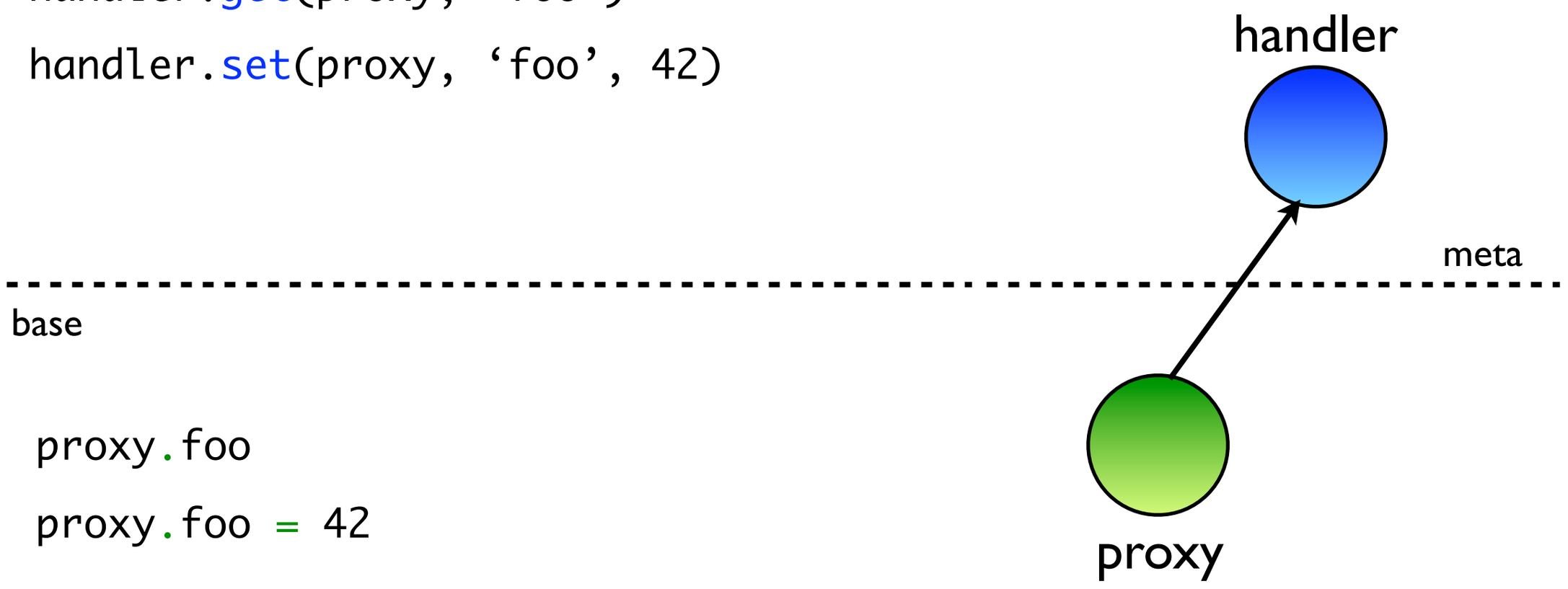
Stratified API

```
var proxy = Proxy(handler);

handler.get(proxy, ‘foo’)
```



Stratified API

```
var proxy = Proxy(handler);

handler.get(proxy, ‘foo’)
handler.set(proxy, ‘foo’, 42)
```

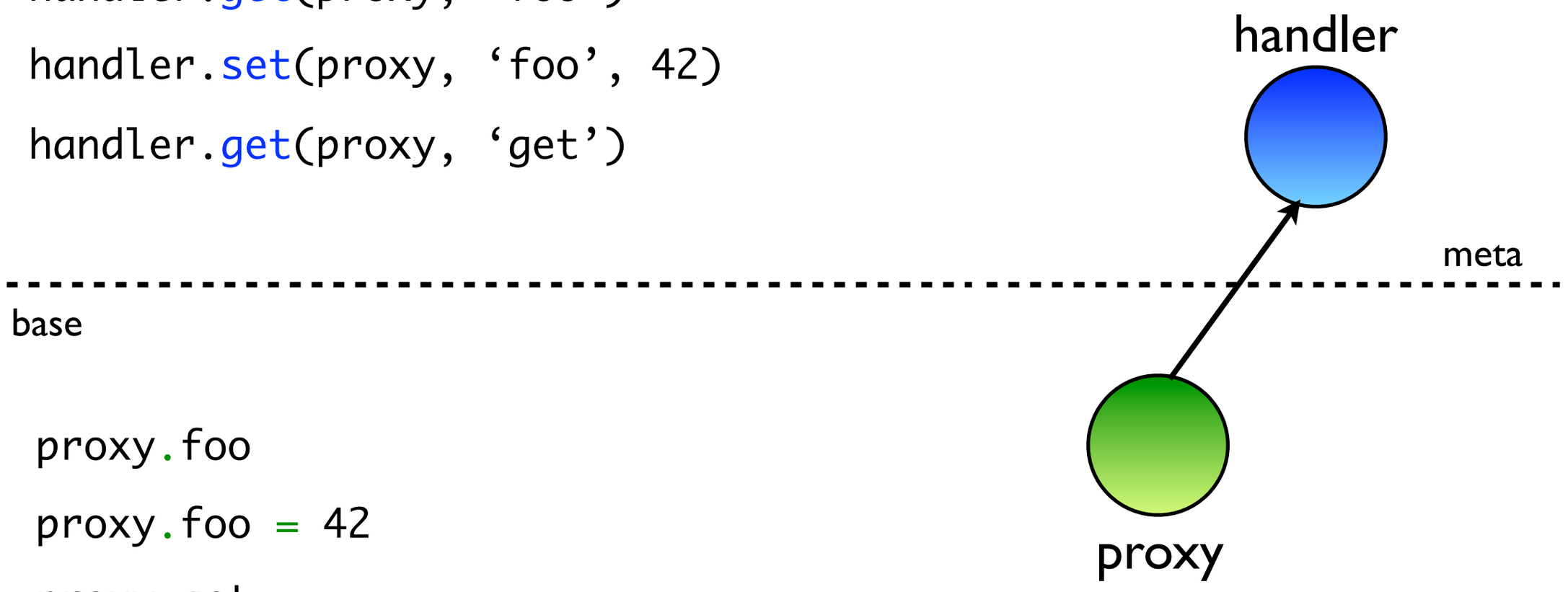



```
proxy.foo
proxy.foo = 42
```

Stratified API

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

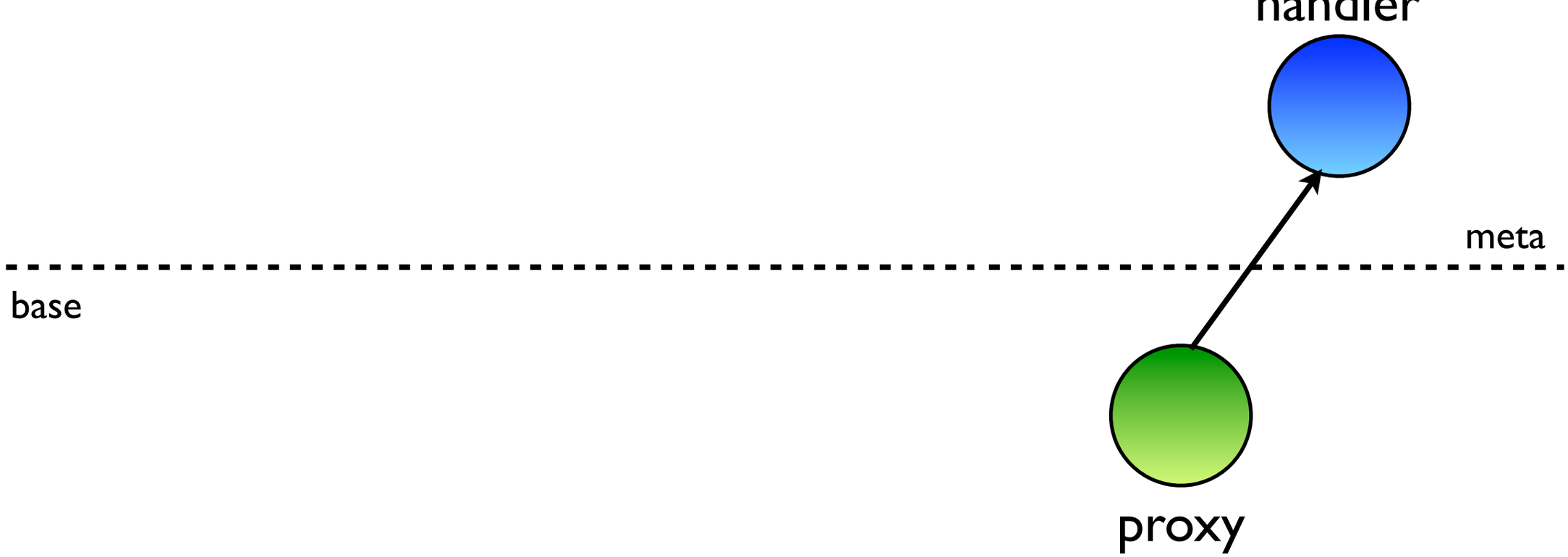
```
handler.get(proxy, ‘foo’)
handler.set(proxy, ‘foo’, 42)
handler.get(proxy, ‘get’)
```




```
proxy.foo
proxy.foo = 42
proxy.get
```

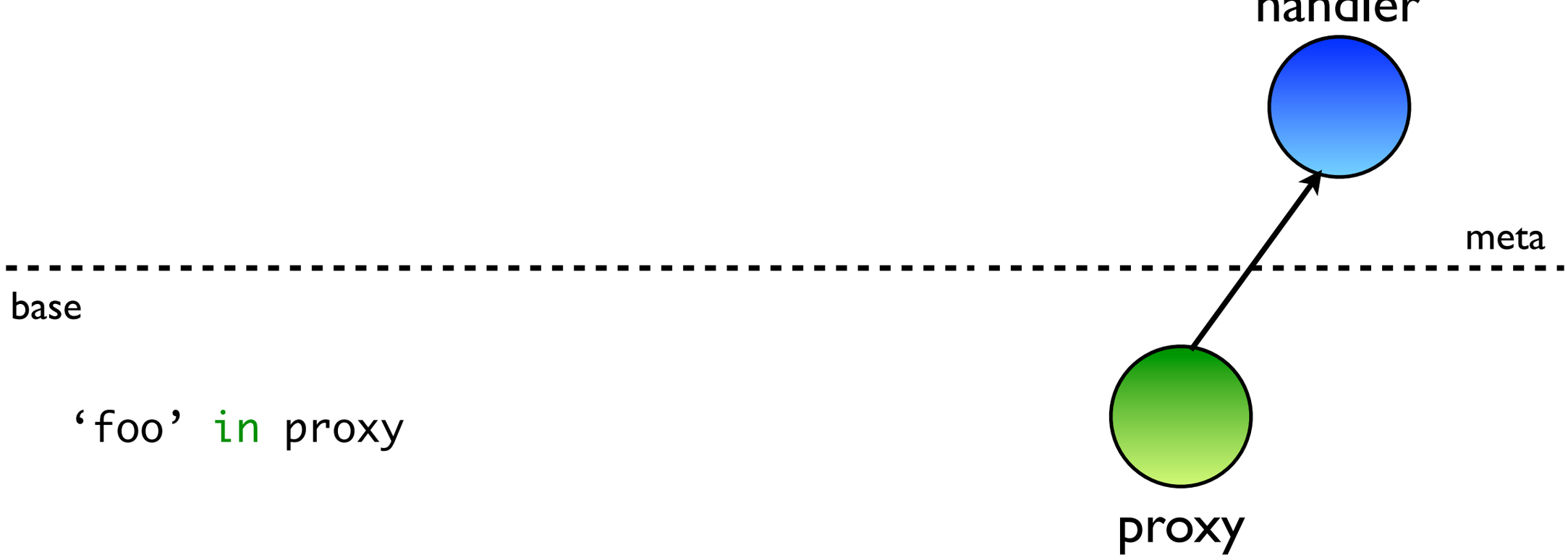
Not just property access...

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



Not just property access...

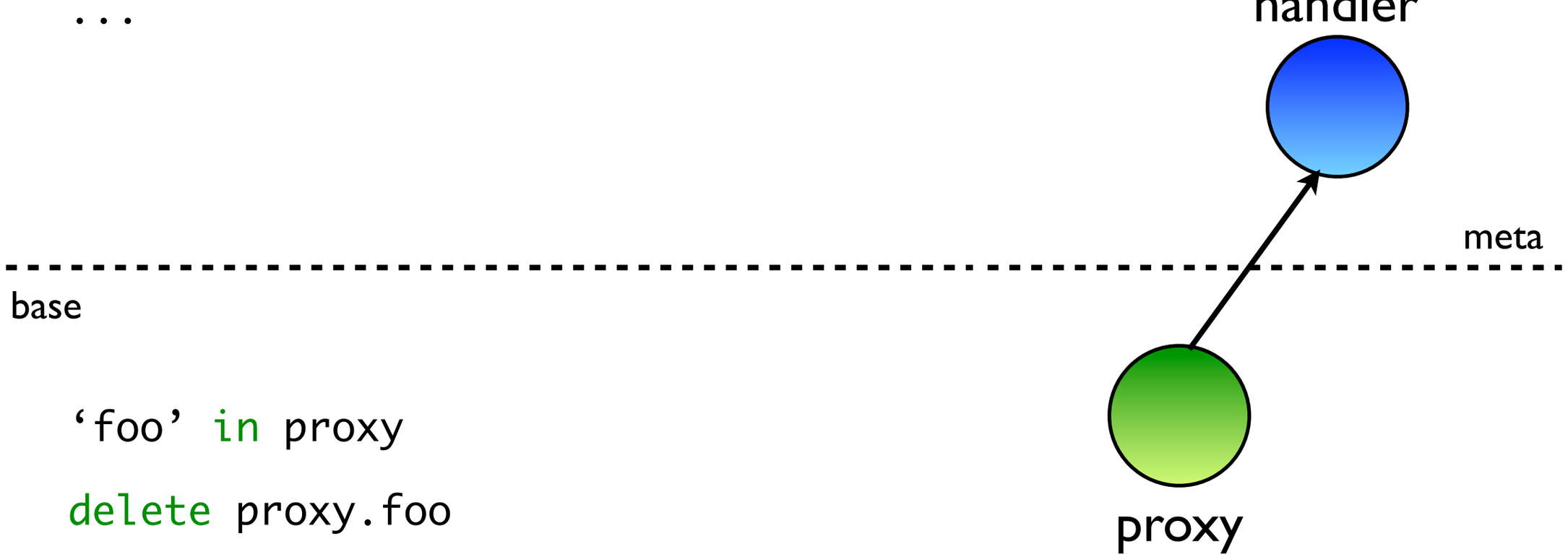
```
var proxy = Proxy(handler);

  handler.has('foo')
```



Not just property access...

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

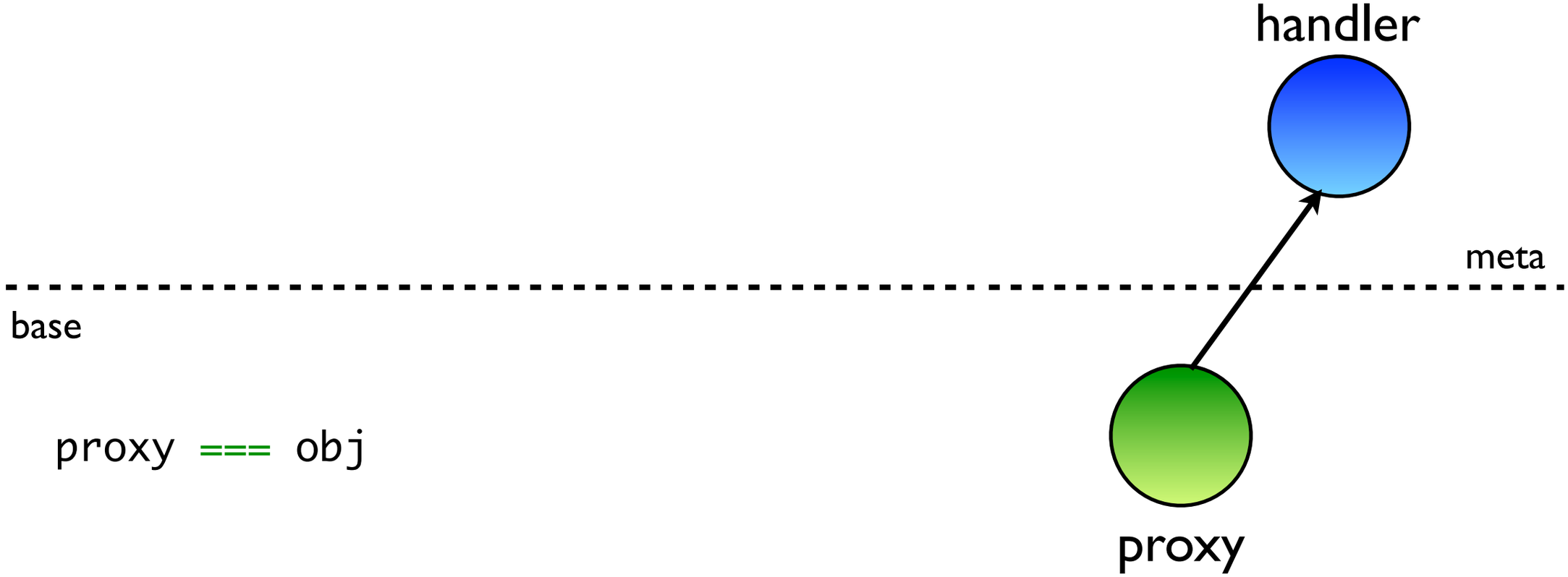
```
  handler.has('foo')

  handler.delete('foo')

  ...
```




```
  'foo' in proxy

  delete proxy.foo

  ...
```

... but not quite everything either

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




```
proxy === obj
```

Frozen objects (new since ECMAScript 5)

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

Object.freeze(point);

point.z = 0;     // error: can't add new properties
delete point.x; // error: can't delete properties

point.x = 7;     // error: can't assign properties

Object.isFrozen(point) // true
```

guarantee (invariant):
properties of a frozen object are immutable

freezing is permanent - there is no **`defrost`**

How to combine proxies with frozen objects?

- Can a proxy emulate the “frozen” invariant of the object it wraps?

```
var point = { x: 0, y: 0 };
Object.freeze(point);
var {proxy,revoke} = makeRevocable(point);

Object.isFrozen(point) // true
Object.isFrozen(proxy) // ?
```

How to combine proxies with frozen objects?

- Can a proxy emulate the “frozen” invariant of the object it wraps?

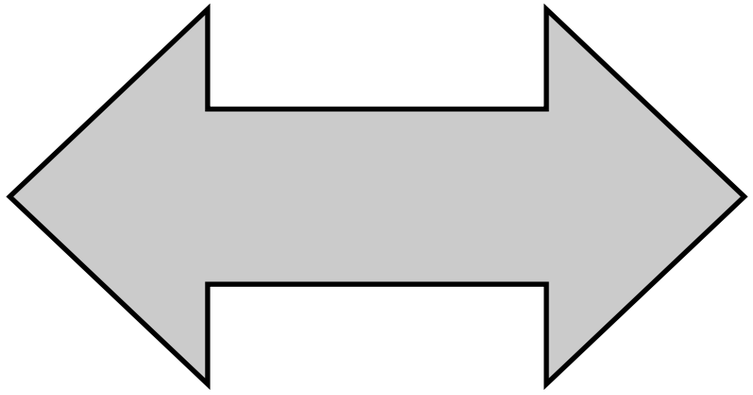
```
function wrap(target) {
  return Proxy({
    get: function(rcvr, name) { return Math.random(); }
  });
}

var point = { x: 0, y: 0 };
Object.freeze(point);

var proxy = wrap(point);

Object.isFrozen(point) // true
Object.isFrozen(proxy) // can’t be true!
```

The “Solution”

- Proxies can’t emulate frozen objects
- `Object.isFrozen(proxy)` always returns `false`
- Safe, but overly restrictive

Language Design Tradeoff

Powerful proxies that can virtualize frozen objects



Strong language invariants that can’t be spoofed

Second iteration: “direct” proxies

- Proxy now has direct pointer to target: `Proxy(target, handler)`
- `Object.isFrozen(proxy)` <=> `Object.isFrozen(target)`

Revocable references (old API)

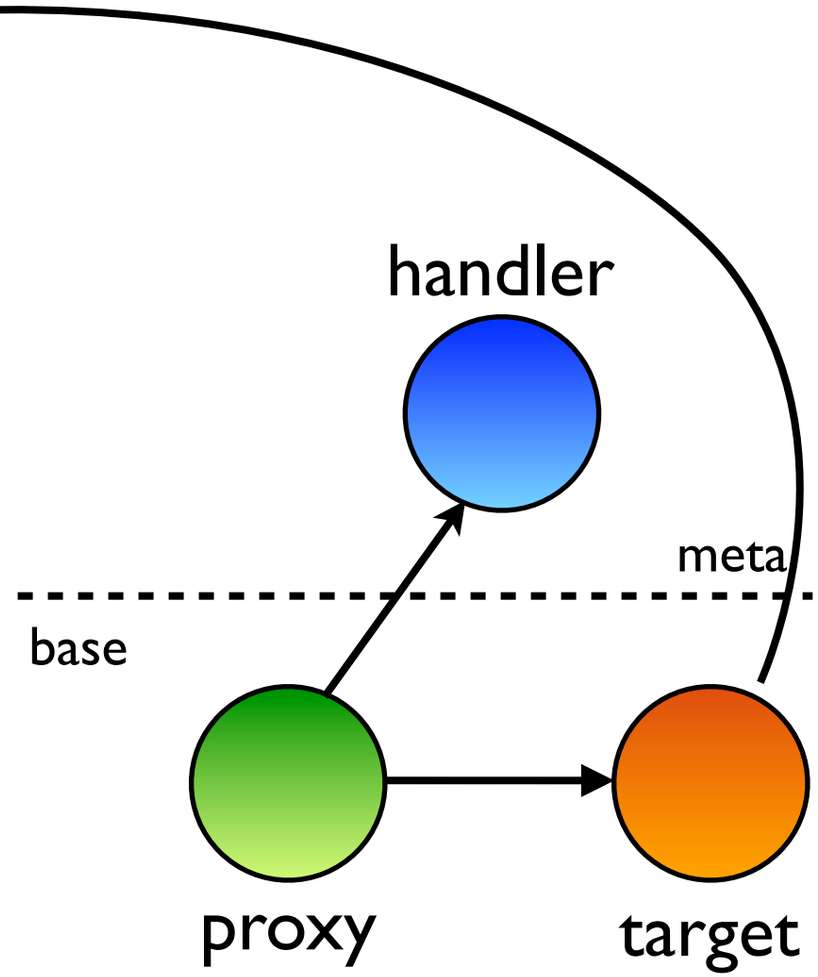
```
function makeRevocable(target) {
  var enabled = true;
  var proxy = Proxy({
    get: function(rcvr, name) {
      if (!enabled) throw Error(“revoked”)
      return target[name];
    },
    set: function(rcvr, name, val) {
      if (!enabled) throw Error(“revoked”)
      target[name] = val;
    },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

Revocable references (new API)

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

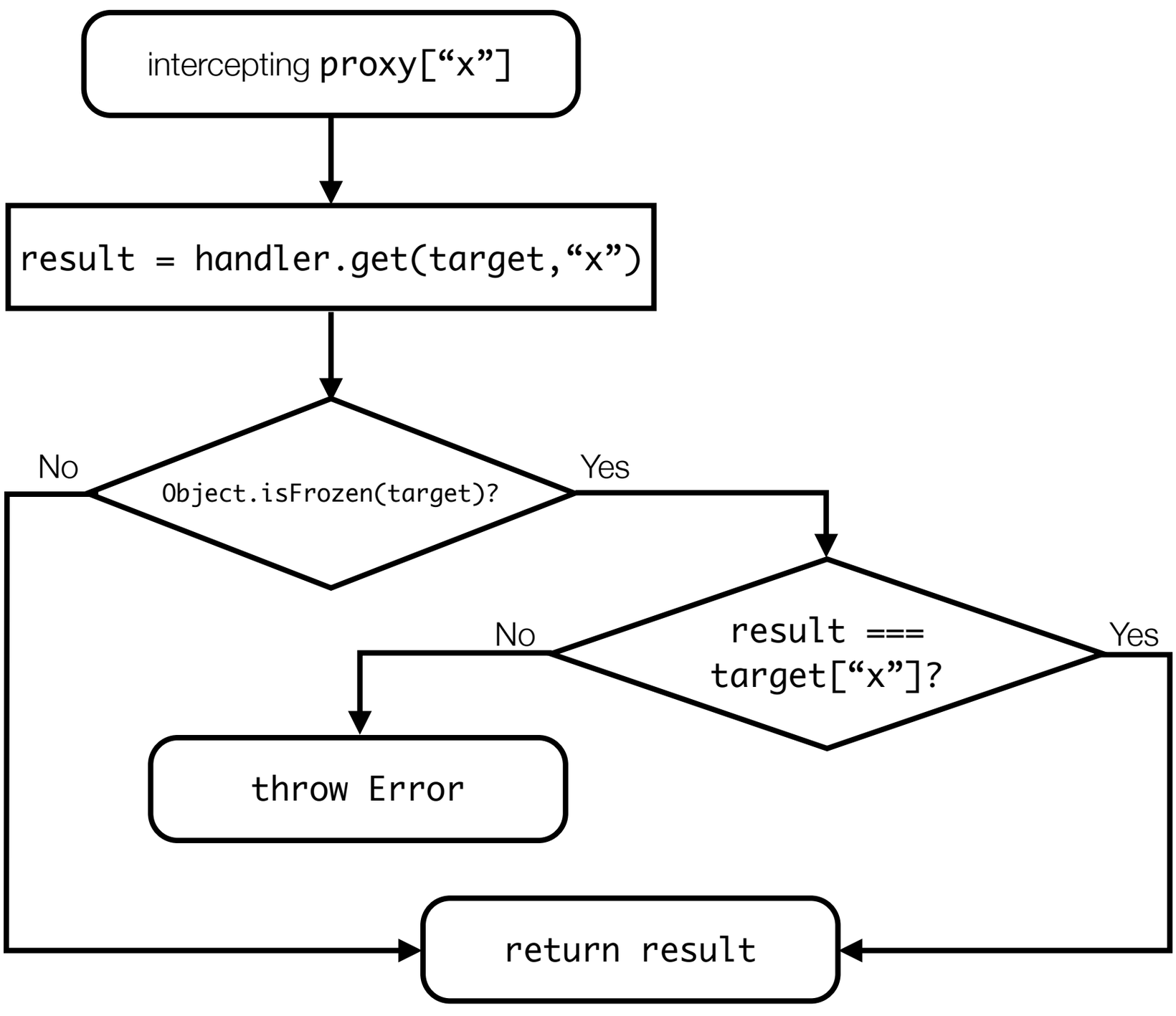
```
function makeRevocable(target) {
  var enabled = true;
  var proxy = Proxy(target, {
    get: function(tgt, name) {
      if (!enabled) throw Error("revoked")
      return target[name];
    },
    set: function(tgt, name, val) {
      if (!enabled) throw Error("revoked")
      target[name] = val;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

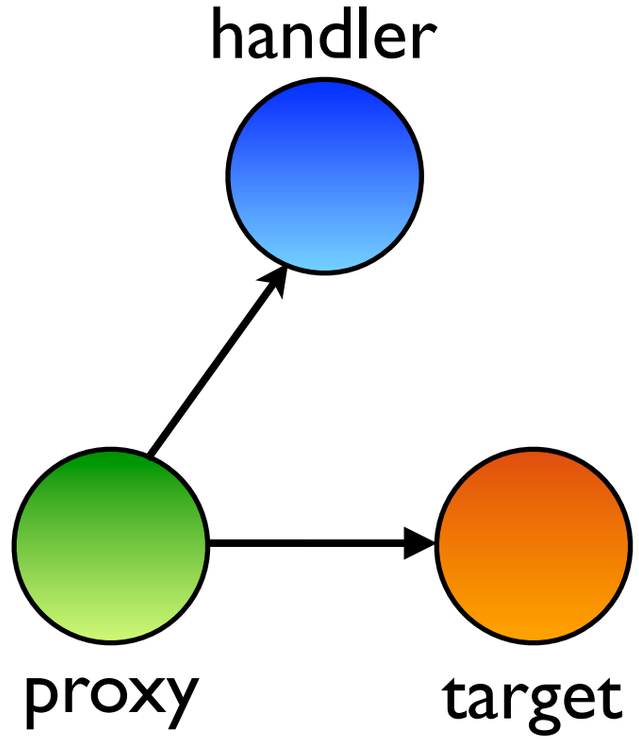


Direct proxies

```
var point = { x: 0, y: 0 };
Object.freeze(point);

var {proxy,revoke} = makeRevocable(point);

Object.isFrozen(point) // true
Object.isFrozen(proxy) // true!
```

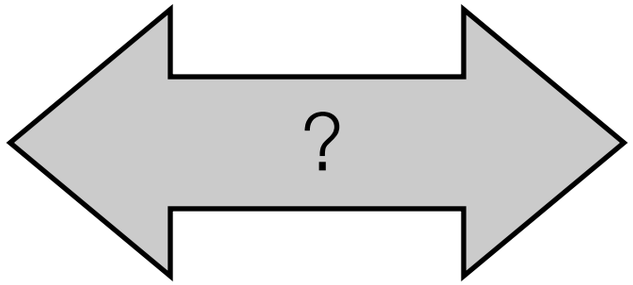
Proxies enforce invariants via runtime assertions



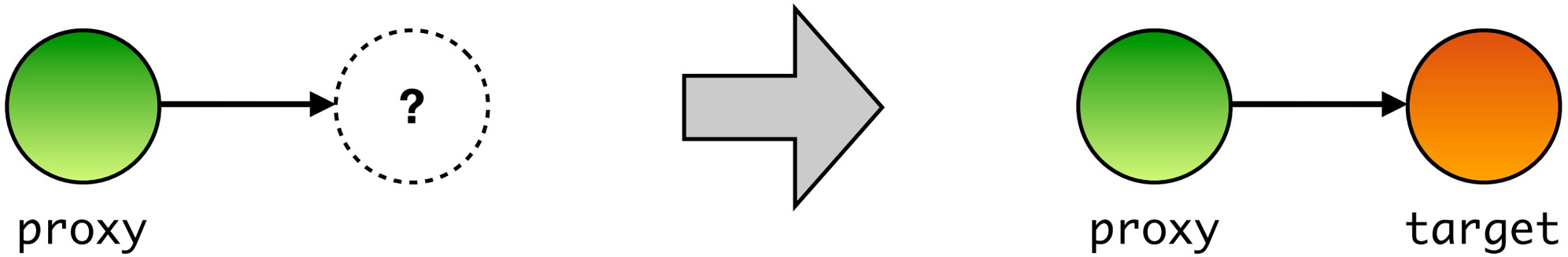


Summary: tradeoffs in language design

Powerful proxies



Strong language invariants



- No free lunch:
  - Direct proxies are more complicated (invariant checks)
  - The two Proxy APIs support dual use cases. But: having *both* virtual and direct proxies in the language further increases complexity.