Ecma/TC39/2010/007

Proxies
Strawman Proposal

Tom Van Cutsem
Mark S. Miller

Google™

Goal

- Enable implementation of virtualized objects ("proxies")
- Enables:
  - Generic abstractions to enforce access control
  - Virtualized objects: persistent, remote, lazily instantiated
  - Virtualizing legacy APIs (adaptors)
  - Transparent logging, tracing, profiling
  - DSLs in property names (cf. ORM frameworks)
  - Intercepting missing method calls (__noSuchMethod__)
  - Higher-order messages
- Synergy with Ephemeron Tables
  - Identity-preserving Membranes

Proposed API

Example: a simple generic forwarding proxy

```
function makeHandler(obj) {
  return {
    has: function(name) { return name in obj; },
    get: function(rcvr,name) { return obj[name]; },
    set: function(rcvr,name,val) { obj[name]=val; return true; },
    invoke: function(rcvr,name,args) { return obj[name](...args); },
    enumerate: function() {
      var res = []; for (name in obj) { res.push(name); }; return res;
    },
    delete: function(name) { return delete obj[name]; }
  };
}

var proxy = Proxy.create(makeHandler(o),
                         Object.getPrototypeOf(o));
```

Architecture

- Stratification: "meta-level" separated from "base-level"
  - No conflation between "traps" and plain methods
  - Only proxy objects can be trapped
  - Handler has independent prototype chain
  - No default access to a proxy's handler

```
var proxy = Proxy.create(handler, proto);
proxy.foo; // handler.get(proxy,'foo');
proxy.get(proxy, 'foo'); // handler.invoke(proxy,'get',[proxy,'foo']);
```

- Security: no trapping on arbitrary existing objects
- Efficiency: no overhead for non-trapped objects
- Careful choice of what should be virtualized:
  - Most "base-level" operations on objects can be trapped

Function proxies

- Spec. distinguishes between Functions and Objects
- Object proxies vs. Function proxies

```
function wrap(f) {
  function callTrap(...args) { return f.call(this, ...args); }
  function constructTrap(...args) { return new f(...args); }
  return Proxy.createFunction(makeHandler(f), callTrap, constructTrap);
}
var fproxy = wrap(function (x,y) { return x + y });
fproxy(1,2); // callTrap.call(this,1,2);
new fproxy(1,2); // constructTrap(1,2);
Object.getPrototypeOf(fproxy) === Function.prototype
typeof fproxy // "function"
f.foo // handler.get(f, 'foo')
```

Fundamental vs. Derived Traps

Fundamental traps

Object.getOwnProperty(proxy)	**getOwnProperty:** function(name) -> pd | undefined
Object.getProperty(proxy)	**getProperty:** function(name) -> pd | undefined
Object.defineProperty(proxy,name,pd)	**defineOwnProperty:** function(name, pd) -> boolean
delete proxy.name	**delete:** function(name) -> boolean
Object.getOwnPropertyNames(proxy)	**getOwnPropertyNames:** function() -> [string]
for (name in proxy)	**enumerate:** function() -> [string]
Object.{freeze|seal|preventExtensions}(proxy)	**fix:** function() -> propertyMap | undefined

Derived traps (less allocations)

name in proxy	**has:** function(name) -> boolean
({}).hasOwnProperty.call(proxy, name)	**hasOwn:** function(name) -> boolean
receiver.name	**get:** function(receiver, name) -> any
receiver.name = val	**set:** function(receiver, name, val) -> boolean
receiver.name(...args)	**invoke:** function(receiver, name, args) -> any
Object.keys(proxy)	**enumerateOwn:** function() -> [string]

freeze, seal, preventExtensions

- Design constraint: proxies should not circumvent constraints guaranteed by Object.{freeze|seal|preventExtensions}
- These primitives trigger a handler's **fix** trap:
  - **fix** returns a property descriptor map
    - A new object is generated from this map
    - The proxy becomes 'fixed': it is indistinguishable from the newly generated object and its handler is permanently bypassed
  - **fix** returns undefined
    - The corresponding primitive throws a TypeError
    - The proxy handler continues to trap operations
- `Proxy.isTrapping(proxy) -> boolean`

Example: a simple membrane

```
function makeSimpleMembrane(target) {
  var enabled = true;

  function wrap(wrapped) {
    if (wrapped !== Object(wrapped)) {
      return wrapped; // primitives are passed through unwrapped
    }
    var baseHandler = makeHandler(wrapped); // a generic no-op forwarder
    var revokeHandler = Proxy.create({
      invoke: function(rcvr, name, args) {
        if (!enabled) { throw new Error("disabled"); }
        return wrap(baseHandler[name](...args.map(wrap)));
      }
    });
    return Proxy.create(revokeHandler,
                        wrap(Object.getPrototypeOf(wrapped)));
  }

  var gate = Object.freeze({
    enable:  function() { enabled = true; },
    disable: function() { enabled = false; }
  });

  return Object.freeze({ wrapper: wrap(target), gate: gate });
}
```

Prior work on stratified intercession

- Reflection/meta-programming: CLOS class metaobjects
- Java proxies: java.lang.reflect.Proxy
  - A single "invoke" trap
  - Only works for interface types

```
public interface Foo { ... }
InvocationHandler handler = new InvocationHandler() {
   public Object invoke(Object p, Method m, Object[] args) {...}
};
Foo p = (Foo) Proxy.newProxyInstance(Foo.class.getClassLoader(),
                                     new Class[] { Foo.class },
                                     handler);
```

- AmbientTalk mirages
  - Multiple traps (one per "meta-level" operation)
- E proxies
  - User-defined "eventual references"

Google™

Conclusion

- Proxies enable **virtualized** objects, which in turn enable:
  - Generic access control wrappers
  - persistent objects, remote objects, lazy objects
  - Adaptors (e.g. emulating legacy API)
  - Generic forwarders (cf. __noSuchMethod__)
  - DSLs in property names
- **Stratified** architecture:
  - No conflation between 'traps' and ordinary methods that incidentally have the same name
  - Handler not accessible from proxy
- No traps on existing objects
- Proxies cannot circumvent the guarantees provided by freeze, seal and preventExtensions