

Hidden Fields and Prototypical Inheritance in JavaScript

Ben Newman
CS242: Programming Languages

May 18, 2006

Introduction

JavaScript is arguably the most misunderstood programming language in use today.¹ The natural supposition that it has something to do with Java turns out to be quite mistaken; in fact, though both languages inherit the syntax of C and are interpreted, further similarities are not easy to find. In JavaScript, functions are first-class objects. Functions can return functions built on the fly or accept other functions as arguments. Almost any kind of property can be added to any object at any time, and object properties can be inspected at run-time. JavaScript is weakly-typed in the strongest sense of the term: virtually every type conversion is implicit. JavaScript is lexically scoped, which makes coding in JavaScript feel more like hacking Lisp than writing Java classes. As a matter of fact, JavaScript and Lisp are so closely related that a compiler *from Lisp to JavaScript* can be implemented in less than 100 lines of code.² One need not be a devotee of the Lambda Calculus to find such a possibility fascinating.

Though JavaScript must be called object-oriented, its model for inheritance is quite different from the classical system used in Java. Java, however, seems to have gotten object orientation right, if there is any conclusion to be drawn from the evidence that it is overwhelmingly the language of choice for demonstrating the principles of object-oriented design in the academy and beyond. I have no intention of disputing the claim that it wears this badge deservedly. The designers of any language which supports object orientation would do well to ponder what they have done differently from the Java engineers and, if the divergence is large, take care to justify their choices with respect to the intended use of their language.

But what exactly is “right” about Java’s object orientation model? One answer is its sobriety. Java is suited for developing large-scale projects because it is rigid enough that assumptions can be enforced and conventions kept. The protocols according to which objects interact can be defined very narrowly, and class definitions can be trusted for the duration of a program. The ability to add a method to an object at run-time in JavaScript may open new doors for clever code, but the abuse of such features threatens the stability of any long-term software project, since cleverness is simply not of the essence when code must be maintainable.

This paper will examine JavaScript’s approach to inheritance, argue that this approach is not flexible enough to be truly innovative but too flexible to

¹Douglas Crockford has written a brief yet thorough essay expounding this claim: <http://www.crockford.com/javascript/javascript.html>.

²See <http://www.cryptopunk.com/wip/lisptojavascript.html>.

be practical, and show that an inheritance mechanism much closer to that of Java can be implemented, with some trouble, using the core of the JavaScript language. After I implemented this richer inheritance mechanism during the Summer, my own work on a web application for group collaboration was greatly expedited, so I will be writing with the aid of direct experience.

JavaScript and Prototypical Inheritance

When it comes to inheritance, JavaScript seems to have taken a cue from the language Self. In Self, objects inherit from so-called “parent” objects; that is, each object has a pointer to its parent (potentially null), and when a method that an object does not support is invoked against the object, a dynamic lookup occurs, following the chain of parents until the method is found or the chain ends in a null pointer.³ In a sense, therefore, an inheritance chain in Self is just another data structure. In fact, since Self permits multiple inheritance, an inheritance hierarchy can even be used as an n -ary search tree.⁴ JavaScript weakens this model by imposing a few additional restrictions:

- The prototype (parent) of an object must be set before the object is constructed (more on this later) and cannot be changed afterwards.
- An object can have at most one prototype. This obviously inhibits multiple inheritance.
- The programmer does not have direct access to an object’s prototype; instead, the language handles dynamic lookups behind the scenes, so that a child object really seems to have every property defined anywhere in its prototype chain.

One of the chief benefits to Self’s inheritance system is that objects can change deep aspects of their behavior with relatively little code. For instance, if the visual display of an abstract window object depends only on its parent, then it can simply swap in different parent objects to display itself in a maximized, minimized, or draggable mode.⁵ This sort of flexibility would require a considerably more complicated GUI infrastructure if it were implemented using Java. If the designers of JavaScript truly had Self in

³Here and elsewhere in the paper, I restrict my discussion to method lookup, although it is worth noting that many of the same remarks apply to data member lookup.

⁴See <http://research.sun.com/techrep/1994/smlr-tr-94-30.pdf>, pages 15-16.

⁵Thanks to Mitchell’s book for that example.

mind when they were designing the language’s inheritance mechanism, they must have deliberately decided that the advantages of this feature were not worthwhile, since JavaScript denies the programmer the ability to change an object’s prototype, even though affording such an ability would seem easy.

One possible explanation for this choice is that, by requiring an object’s prototype to be fixed before it is constructed, JavaScript creates a stable notion of subtyping for its objects. Since the type of an object is defined by its entire inheritance chain, and since this chain is subject to change in Self, identifying the type of an objects in Self is a potentially costly computation. In JavaScript and Java, by contrast, the types of objects are in some sense fixed at compile time (“instantiation time” for JavaScript). Given such knowledge, a programmer can make stronger assumptions about the use of objects in the program, which greatly simplifies the task of program verification. In Java’s case, the assumption of a fixed inheritance chain allows static type analysis at compile time. In JavaScript, the advantage is more dependent upon the programmer’s diligence, but it is a considerable advantage nevertheless, especially if several people are working on the same code and must constantly reconcile their assumptions about the interdependent parts of the program.

It should now be clear that while the implementation of inheritance in JavaScript mirrors that of Self, JavaScript forfeits some of the key features of Self and, in so doing, moves nearer to the classical paradigm exemplified by Java. Unfortunately, JavaScript fails to incorporate several of the vital features that have made classical inheritance so successful.

Limitations of Inheritance in JavaScript

Though I have titled this section in the plural, I will spend the rest of the paper describing a solution to only one limitation of JavaScript’s inheritance scheme. That limitation is the problem of *hidden fields*. Whenever a subclass overrides a method in a superclass, the superclass version of the method will no longer be accessible to objects created from the subclass, unless some mechanism is provided that grants explicit access to superclass methods. In Java, the `super` keyword provides such access. JavaScript, however, has no such mechanism.

Before proceeding further, let it be clear that access to hidden fields is a feature at the heart of classical inheritance. The ability to override methods in such a way that the previous version of the method can be leveraged in the new version often allows massive savings in lines of code

required to implement a subclass and in the amount of memory required to represent an object instantiated from the subclass. The code reuse issue is particularly pertinent for JavaScript, since the language is most widely used as a scripting tool for client-side web interaction. Given that the code itself must be transported over the network before being interpreted on the user's computer, minimizing the quantity of code is a significant desire. In a more informal sense, having any experience programming in a language which affords selective access to hidden fields makes it quite tiresome to program in a language that denies that ability.

Since JavaScript allows the removal of methods from objects at run-time, one possible implementation of a `super`-like keyword would be to remove the desired method from the current object and then invoke the method against the object as usual, so that the prototype chain will have to be searched for a matching method. This simple approach turns out to be doomed for a number of reasons. Most immediately, if the method was inherited from a superclass, since the programmer cannot directly access the prototype of an object, it is impossible for the programmer to truly remove the method from the object's prototype. The most s/he can do is to create a property in the base object with the same name and set it equal to `undefined`. But this clearly does not have the desired effect: we want an earlier version of the method, but instead we are told that the method no longer exists anywhere in the inheritance hierarchy.

The solution I will describe during the rest of this paper succeeds in at least five respects:

- It makes full use of the dynamic lookup mechanism already present in the language (i.e. there's no need to rebuild that functionality in an inevitably less efficient way).
- It use looks syntactically almost identical to the use of Java's `super` keyword. Since `super` is a reserved word in JavaScript (future plans?), I use `sup` instead. You'll like it.
- The `sup` keyword is implemented once for an entire class, rather than individually for each method.
- Nested `sup` calls work as they should.
- The ability to use `sup` is granted on an opt-in basis. In fact, built-in JavaScript objects (which of course do not make use of `sup`) can be used as superclasses without problem.

The Scheme Unveiled

I took initial inspiration from a library called Prototype.js⁶, which is being used as the core of the new AJAX⁷ features in the Ruby on Rails web-development infrastructure. This library exposes an object called `Class`, which has one method, `create`. The purpose of the code is quite simple. The syntax may be new to you, but I will explain the important bits immediately afterwards.

```
var Class = {
  create: function() {
    return function() {
      this.initialize.apply(this, arguments);
    }
  }
}
```

The syntax `function (arguments) {body}` is analogous to `(lambda (arguments) body)` in Lisp. An object literal is represented as an associative array enclosed by curly braces; that is

```
{key: value, key: value, ...}
```

In the above code, there is only one key, `create`, and its value is a function expression. This function returns another function which applies the `initialize` method of the object to any arguments that were passed into the function. In JavaScript, the `arguments` array is available inside any function as an alternative means of accessing the function's arguments in an unnamed way (i.e. just by index). The first argument to `this.initialize.apply` sets the value of the `this` pointer that will be available inside `initialize` to be the same as the `this` pointer within the returned function.

The `Class` object is intended to be used in the following manner:

```
var MyClass = Class.create();
MyClass.prototype = {
  initialize: function(val) {
    this.value = this.addOne(val);
  },
}
```

⁶See <http://prototype.conio.net/>. I've gotten in touch with the creators of the library since developing my inheritance scheme, and they seem excited by the possibilities it promises (for instance, automatic documentation generation).

⁷If you're not familiar with this term, see http://en.wikipedia.org/wiki/Ajax_%28programming%29.

```

    addOne: function(num) {
        return num + 1;
    }
}

var obj = new MyClass(4);
// at this point, obj.value equals 5

```

Note that by setting the `prototype` of `MyClass`, which is a function that we're using as a constructor, we are defining the object that will be used as the prototype for all objects instantiated by evaluating the `new MyClass(val)` expression. This makes some of my earlier claims more concrete: the prototype of the constructor function has to be set before the `new` keyword is used to create an object. Think of the constructor function simply as a function that is executed in a context where `this` points to a newly-allocated block of memory.

The only purpose of the `Class.create` code is to allow the client to specify that `initialize` should be invoked against the newly instantiated object. To a Java programmer, `initialize` seems like a constructor, and that is its intuitive purpose, but technically the true constructor is the function returned by `Class.create()`, which happens to call `this.initialize`.

On the surface, my modification makes a slight simplification of the conventions just demonstrated. Instead of the two-step class definition employed above, one would write

```

var MyClass = Class.create({
    initialize: function(val) {
        this.value = this.addOne(val);
    },
    addOne: function(num) {
        return num + 1;
    }
});

var obj = new MyClass(4);
// obj.value equals 5, as before

```

When the new interface is used, however, a property named `extending` may be included in the hash-map to denote an inheritance relationship (note that `Class.create` now takes a single argument that is an object literal). One would use this facility in the following manner:

```

var MyClass = Class.create({
  initialize: function(val) {
    this.value = this.addOne(val);
  },
  addOne: function(num) {
    return num + 1;
  }
});

var MySubClass = Class.create({
  extending: MyClass,
  initialize: function(num) {
    this.value = 2 * this.addOne(num);
  }
});

var obj = new MySubClass(4);
// obj.value equals 10

```

As you can probably guess, `Class.create` has essentially become a factory method for setting up the prototype chain in the appropriate manner. The necessary code would be somewhat hairy to write by hand, so an interface like the one my scheme affords saves some typing (and some thinking).

The real virtue of the scheme, however, is that the function `this.sup` gets defined on the client's behalf and is subsequently available inside any of the class methods. So we could have written `MySubClass` thus:

```

var MySubClass = Class.create({
  extending: MyClass,
  initialize: function(num) {
    this.sup(num);
    this.value *= 2;
  }
});

```

The savings here aren't very substantial, but if the `initialize` method of `MyClass` were responsible for more operations, `this.sup` would become quite handy.

The details of implementing all of this would take more explanation than the length of this paper permits, so I will explain only the most interesting part: the creation of the `sup` method. The entirety of the code, with ample comments, will be included at the end of the paper, for your optional perusal.

Implementing `Class.makeSuper`

In addition to `initialize`, my `Class` object implements a number of private helper methods. One of these methods, called `makeSuper`, returns a function that can be called within any method of the class in order to invoke an overridden version of the method. This function gets assigned to the object as the property named `sup`. The `makeSuper` method is implemented according to the following logic:

```
makeSuper: function() {
  // where to start lookup; default to current object
  var __platform = false;

  return function() {
    var method = __nameOfLastCalledMethod();
    var curObj = __platform || this;

    // walk up inheritance chain until we've found the first class that
    // actually implements the method from which this.sup was called
    while ( curObj.ownPrototypeHas &&
            !curObj.ownPrototypeHas(method) &&
            curObj.getBaseInstance()
            curObj = curObj.getBaseInstance());

    // now go one step further, so that curObj[method] refers to the
    // desired property of the true superclass
    curObj = curObj.getBaseInstance
      ? curObj.getBaseInstance()
      : {}; // the ultimate baseInstance

    var previousBase = __platform;
    __platform = curObj; // allow nested this.sup() calls
    try {
      var result = curObj[method].apply(this, arguments);
    } catch (exception) {
      throw exception;
    } finally { __platform = previousBase; }
    return result;
  };
};
```

The reason we have to walk up the inheritance chain is so that our search for the superclass begins at the class which actually implemented the method in which `this.sup` was called. If we just begin our search with the object returned by `this.getBaseInstance()`, the first method with the desired signature we encounter might be the same method from which `this.sup` was called, which would cause an infinite loop.

In order to implement `__nameOfLastCalledMethod`, I had to maintain my own call stack, which required wrapping each class method in another method which pushed the method's name onto the stack, called the method, and then popped its name off the stack. Another way of accomplishing the same thing would be to examine `arguments.callee.caller` to determine the calling function, but, alas, Opera and Safari don't fully support that part of the ECMA-262 standard⁸. The code included at the end of the paper works in any modern browser.

Conclusion

My work on this material has consisted of a long series of show-stopping dead-ends that somehow ceased to be problematic after a day or two of hard thought. The result is a system that I can honestly say makes my JavaScript hacking more enjoyable. My task might have been simplified if I had been able to modify one of the existing JavaScript interpreters, but that was clearly not an option, because there are few things more difficult than getting everyone to use a new browser. The library-based approach I have taken seems the only viable option.

The code included below implements several features that were not mentioned in the paper; for instance, the run-time addition of new methods that are `sup`-aware, and the enforcement of privacy restrictions. I encourage you to read through it if you have the slightest interest in JavaScript as a language.

This paper is a side-effect of work that motivated me to take CS242 in the first place. I don't think I would have any trouble filling another fifty pages with a complete rationale for my work. I have only hoped to give a taste of its current state.

⁸See <http://www.ecma-international.org/publications/standards/Ecma-262.htm>.

The Whole Shebang

```
var Class = new (function() {

    // You'll definitely want to check out
    // http://groupspace.org/devben/proto-svn/demo/
    // to get a mile-high understanding of why I
    // bothered with all of this.

    this.NAME = "Class";
    this.VERSION = Prototype.Version || 2.0;
    // put your name here if you've helped in any way!
    this.AUTHORS = "Ben Newman (of groupspace.org), et al.";

    this.__repr__ = function() {
        return "[" + this.NAME + " " + this.VERSION + "];";
    };

    this.toString = function() {
        return this.__repr__();
    };

    var __doNotInit = {};

    this.create = function(declarations) {

        var ctor = function() {
            if (arguments[0] !== __doNotInit) {
                this._runTimeDeclarations = {};
                if (this.initialize)
                    this.initialize.apply(this, arguments);
            }
        };

        if (declarations) {
            // runs when we call __inherit(ctor, ownPrototype) below
            var ownPrototype = function() {
                for (var i in declarations)
                    __attachClassProperty(i, declarations[i], this);
                __ensureDefaults(this, declarations);
            };
        }
    };
});
```

```

    __ensureOwnPrototypeMemory(this, declarations);
};

// every class extends another class or Object
var base = declarations.extending || {};

// baseInstance is the prototype *of* our prototype
var baseInstance = __inherit(ownPrototype, base);
declarations['getBaseInstance'] = Class.encap(baseInstance);
declarations['sup'] = __makeSup();

__markCriticalFunctions(declarations);
__setupAttachDelete(declarations);
__inherit(ctor, ownPrototype);
}

// see http://groupspace.org/devben/proto-svn/demo for examples of
// the appropriate use of this experimental feature:
ctor['mimic'] = function(obj) {
    // Unfortunately it seems that you just can't use
    // DOM objects as prototypes for constructor functions
    // in Safari... how braindead is that? Ugh :(
    var __ctor = this;
    var fn = function(/*arguments*/) {
        __ctor.apply(this, arguments);
    };
    Object.extend(obj, __ctor.prototype);
    __ensureDefaults(obj, __ctor.prototype);
    fn.prototype = obj;
    return fn;
};

return ctor;
};

// superclass can be constructor function or any existing object
var __inherit = function(subclass, superclass) {
    if (typeof(superclass) == 'function') {
        superclass.getInstance = superclass.getInstance ||
            Class.encap(new superclass(__doNotInit));
    }
};

```

```

        subclass.prototype = superclass.getInstance();
    } else if (typeof(superclass) == 'object') {
        subclass.prototype = superclass;
        superclass.initialize = superclass.constructor;
    }
    return subclass.prototype;
};

// Functions marked here will be denied this.sup and privacy benefits,
// but will be slightly faster because they will be called directly.
var __markCriticalFunctions = function(declarations) {
    var critical = declarations.critical;
    if (typeof(critical) == 'object') {
        for (var i in critical) {
            critical[i]._critical = true;
            declarations[i] = critical[i];
        }
    }
};

var __setupAttachDelete = function(declarations) {
    // these two methods allow runtime method attachment and deletion,
    // with all the benefits of this.sup and privacy restrictions
    declarations['attachMethod'] = function(fnName, func,
                                           /*optional*/ critical) {
        // can't replace existing methods
        if (!this.ownPrototypeHas(fnName)) {
            func._critical = !!critical;
            this._runTimeDeclarations[fnName] = true;
            __attachClassProperty(fnName, func, this);
        }
    };
    declarations['deleteMethod'] = function(fnName) {
        // can only delete non-runtime additions
        if (this._runTimeDeclarations[fnName]) {
            __attachClassProperty(fnName, null, this, true);
            this._runTimeDeclarations[fnName] = false;
        }
    };
};

```

```

var __attachClassProperty = function(propName, prop, obj,
                                     /*optional*/ removeInstead) {
  if (removeInstead) {
    if (obj.ownPrototypeHas(propName)) delete obj[propName];
  } else {
    obj[propName] = prop;
    if (typeof(prop) == 'function') {
      if (!prop._critical && typeof(prop.body) != 'function')
        __addStackLogic(obj, propName);
      var __origStr = prop.toString(); // never changes
      prop.toString = obj[propName].toString = function() {
        return __origStr.replace(/^s*function/, propName);
      };
    }
  }
};

// special-purpose array object (slightly faster in tests)
var __callStack = new (function() {
  var stack = []; var index = 0;
  this.push = function(what) { stack[index++] = what; }
  this.pop = function() { return stack[--index]; }
  this.empty = function() { return index == 0; }
  this.peek = function() { return stack[index - 1]; }
})();

var __lastOnCallStack = function() {
  if (__callStack.empty()) return ['', null];
  else return __callStack.peek();
};

var __addStackLogic = function(obj, fnName) {
  switch (fnName) {
    case 'sup': case 'valueOf':
    case 'attachMethod': case 'deleteMethod':
    case 'getBaseInstance': return;
    default: {
      var __func = obj[fnName];
      obj[fnName] = function() {

```

```

    // this privacy check can be omitted for slight performance
    // gains in production code (privacy is a development tool)
    var lastRef = __refOfLastCallingObject();
    if (Class.followsPrivateNamingConvention(fnName) &&
        (!lastRef || lastRef.constructor !== this.constructor)) {
        Class.privateCallError(fnName);
    } else {
        __callStack.push([fnName, this]);
        try { // don't let exceptions corrupt the callstack
            var result = __func.apply(this, arguments);
        } catch (exception) {
            throw exception;
        } finally {
            __callStack.pop();
        }
        return result;
    }
};

}
obj[fnName].body = __func; // preserve original
// e.g. obj[fnName].body.call(obj, arg1, ...)
}
};

Function.prototype.grant = function(obj) {
    var __func = this;
    var __callStackMemory = __lastOnCallStack();
    return function() {
        __callStack.push(__callStackMemory);
        try { // don't let exceptions corrupt the callstack
            var result = __func.apply(obj || this, arguments);
        } catch (exception) {
            throw exception;
        } finally {
            __callStack.pop();
        }
        return result;
    }
};

```

```

// consider exposing these (they expose nothing but useful info)
var __nameOfLastCalledMethod = function() {
    return __lastOnCallStack()[0];
};

var __refOfLastCallingObject = function() {
    return __lastOnCallStack()[1];
};

// change this at will
this.followsPrivateNamingConvention = function(propName) {
    return propName.charAt(0) == '_';
};

this.privateCallError = function(fnName) {
    alert("Error: Tried to access private member " + fnName +
        " from from non-member method " +
        __nameOfLastCalledMethod());
};

var __ensureDefaults = function(ownPrototype, declarations) {
    ['toString', 'toLocaleString', 'valueOf'].each(function(prop) {
        __attachClassProperty(prop, declarations[prop], ownPrototype);
    });
};

var __ensureOwnPrototypeMemory = function(ownPrototype, declarations) {
    ownPrototype.ownPrototypeHas = function(property) {
        return (typeof(declarations[property]) != 'undefined' ||
            (this._runTimeDeclarations && // present if instantiated
            this._runTimeDeclarations[property]));
    };
};

// convenient mechanism for creating accessor functions
this.encap = function(__val, mutable) {
    if (!mutable) return function() { return __val; }
    else return function(/* optional new value */) {
        if (arguments.length > 0) __val = arguments[0];
        return __val;
    };
};

```

```

    }
};

var __makeSup = function() {
    var __platform = false; // where to start lookup

    return function() {
        var method = __nameOfLastCalledMethod();
        var curObj = __platform || this;

        // walk up inheritance chain until we've found the first class that
        // actually implements the method from which this.sup was called
        while ( curObj.ownPrototypeHas &&
                !curObj.ownPrototypeHas(method) &&
                curObj.getBaseInstance()
                curObj = curObj.getBaseInstance();
        curObj = curObj.getBaseInstance
        ? curObj.getBaseInstance()
        : {}; // the ultimate baseInstance

        var prevBase = __platform;
        __platform = curObj;
        try { // don't let exceptions break the lookup
            var result = curObj[method].apply(this, arguments);
        } catch (exception) {
            throw exception;
        } finally {
            __platform = prevBase;
        }
        return result;
    };
};

}))(); // good place to pass in parameters that determine what features are
// enabled (production, development, debug, etc.)

var createClass = Class.create;

```