



RELATED READING

Secrets of the JavaScript Ninja
John Resig

MEAP Release: March 2008
Paper book: January 2009 (est.) | 300 pages
ISBN: 193398869X

*This article is excerpted from the upcoming title *The Secrets of JavaScript Ninjas* by John Resig and published by Manning Publications. This article introduces the technique of emulating classical-style inheritance in JavaScript. For the table of contents, source code, the Author Forum, and other resources, go to www.manning.com/resig.*

A common desire, for most developers, is a simplification--or abstraction--of JavaScript's inheritance system into one that they are more familiar with. This tends to head towards the realm of, what a typical developer would consider to be, Classes. While JavaScript doesn't support classical inheritance natively (at least not until JavaScript 2), generally there are a few features that developers crave:

- A system which trivializes the syntax building new constructor functions and prototypes
- An easy way of performing prototype inheritance
- A way of accessing methods overridden by the function's prototype

For example, if you had a system which made this process easier, listing 1 is an example of what you can do with it.

Listing 1: An example of classical-style inheritance, using the code from Listing 2

```
var Person = Object.subClass({
  init: function(isDancing){
    this.dancing = isDancing;
  },
  dance: function(){
    return this.dancing;
  }
});

var Ninja = Person.subClass({
  init: function(){
    this._super( false );
  },
  dance: function(){
    // Call the inherited version of dance()
    return this._super();
  },
  swingSword: function(){
    return true;
  }
});
```

```

var p = new Person(true);
assert( p.dance(), "Method returns the internal true value." );

var n = new Ninja();
assert( n.swingSword(), "Get true, as we expect." );
assert( !n.dance(), "The inverse of the super method's value - false." );

// Should all be true
assert( p instanceof Person && n instanceof Ninja && n instanceof Person,
  "The objects inherit functionality from the correct sources." );

```

A couple things to note about this implementation:

- Creating a constructor had to be simple (in this case simply providing an init method does the trick).
- In order to create a new 'class' you must extend (sub-class) an existing constructor function.
- All of the 'classes' inherit from a single ancestor: Object. Therefore if you want to create a brand new class it must be a sub-class of Object (completely mimicking the current prototype system).
- And the most challenging one: Access to overridden methods had to be provided (with their context properly set). You can see this with the use of this._super(), above, calling the original init() and dance() methods of the Person super-class.

There are a number of different JavaScript classical-inheritance-simulating libraries that already exist. Out of all them there are two that stand up above the others: The implementations within base2 and Prototype. While they contain a number of advanced features the absolute core is the important part of the solution and is what's distilled in the implementation. It helps to enforce the notion of 'classes' as a structure, maintains simple inheritance, and allows for the super method calling, as shown in listing 2.

Listing 2: A simple class creation library

```

// Inspired by base2 and Prototype
(function(){
  var initializing = false,
    // Determine if functions can be serialized
    fnTest = /xyz/.test(function(){xyz;}) ? /\b_super\b/ : /.*/;

  // Create a new Class that inherits from this class
  Object.subClass = function(prop) {
    var _super = this.prototype;

    // Instantiate a base class (but only create the instance,
    // don't run the init constructor)
    initializing = true;
    var proto = new this();
    initializing = false;

    // Copy the properties over onto the new prototype
    for (var name in prop) {
      // Check if we're overwriting an existing function
      proto[name] = typeof prop[name] == "function" &&
        typeof _super[name] == "function" && fnTest.test(prop[name]) ?
        (function(name, fn){
          return function() {
            var tmp = this._super;

            // Add a new ._super() method that is the same method
            // but on the super-class
            this._super = _super[name];

            // The method only need to be bound temporarily, so we
            // remove it when we're done executing
            var ret = fn.apply(this, arguments);
            this._super = tmp;

```

```

        return ret;
    };
    })(name, prop[name]) :
    prop[name];
}

// The dummy class constructor
function Class() {
    // All construction is actually done in the init method
    if ( !initializing && this.init )
        this.init.apply(this, arguments);
}

// Populate our constructed prototype object
Class.prototype = proto;

// Enforce the constructor to be what we expect
Class.constructor = Class;

// And make this class extendable
Class.subClass = arguments.callee;

return Class;
};
})();

```

The two trickiest parts are the "initializing/don't call init" and "create `_super` method" portions. Having a good understanding of what's being achieved in these areas will help with your understanding of the full implementation.

Initialization

In order to simulate inheritance with a function prototype we use the traditional technique of creating an instance of the super-class function and assigning it to the prototype. Without using the above, it would look something like listing 3.

Listing 3: Another example of maintaining the prototype chain

```

function Person(){}
function Ninja(){}
Ninja.prototype = new Person();

// Allows for instanceof to work:
assert( (new Ninja()) instanceof Person, "Ninja is a Person" );

```

What's challenging about this, though, is that all we really want is the benefits of `instanceof`, not the whole cost of instantiating a `Person` object and running its constructor. To counteract this we have a variable in our code, `initializing`, that is set to true whenever we want to instantiate a class with the sole purpose of using it for a prototype.

Thus when it comes time to actually construct the function we make sure that we're not in an initialization mode and run the `init` method accordingly:

```

if ( !initializing )
    this.init.apply(this, arguments);

```

What's especially important about this is that the `init` method could be running all sorts of costly startup code (connecting to a server, creating DOM elements, who knows) so circumventing this ends up working quite well.

Super Method

When you're doing inheritance, creating a class that inherits functionality from a super-class, a frequent desire is the ability to access a method that you've overridden. The final result, in this particular implementation, is a new temporary method (`._super`) which is only accessible from within a sub-classes' method, referencing the super-classes' associated method. For example, if you wanted to call a super-classes' constructor you could do that with the technique shown in listing 4.

Listing 4: An example of calling the inherited super method

```

var Person = Object.subClass({
  init: function(isDancing){
    this.dancing = isDancing;
  }
});

var Ninja = Person.subClass({
  init: function(){
    this._super( false );
  }
});

var p = new Person(true);
assert( p.dancing, "The person is dancing." );

var n = new Ninja();
assert( !n.dancing, "The ninja is never dancing." );

```

Implementing this functionality is a multi-step process. To start, note the object literal that we're using to extend an existing class (such as the one being passed in to `Person.subClass`) needs to be merged on to the base `new Person` instance (the construction of which was described previously). During this merge we do a rather complex check: Is the property that we're attempting to merge a function and is what we're replacing also a function - and does our function contain any use of the `_super` method? If that's the case then we need to go about creating a way for our super method to work.

In order to determine if our function contains the use of a `_super` method we must use a trick provided by most browsers: function decompilation. This occurs when you convert a function to a string, you end up with the contents of the function in a serialized form. We're simply using this as a shortcut so that we won't have to add the extra super-method overhead if we don't have to. In order to determine this we must first see if we can serialize methods properly (currently, only Opera Mobile doesn't do the serialization properly - but it's better to be safe here). That gives us this line:

```
fnTest = /xyz/.test(function(){xyz;}) ? /\b_super\b/ : ./*/
```

All this is doing is determining if the string version of the function contains the variable that we're expecting and, if so, producing a working regular expression and if not, producing a regular expression that'll match any function.

Note that we create an anonymous closure (which returns a function) that will encapsulate the new super-enhanced method. To start we need to be a good citizen and save a reference to the old `this._super` (disregarding if it actually exists) and restore it after we're done. This will help for the case where a variable with the same name already exists (don't want to accidentally blow it away).

Next we create the new `_super` method, which is just a reference to the method that exists on the super-class' prototype. Thankfully we don't have to make any additional changes, or re-scoping, here as the context of the function will be set automatically when it's a property of our object (this will refer to our instance as opposed to the super-class').

Finally we call our original method, it does its work (possibly making use of `_super` as well) after which we restore `_super` to its original state and return from the function.

Now there's a number of ways in which a similar result, to the above, could be achieved (there are implementations that have bound the super method to the method itself, accessible from `arguments.callee`) but this particular technique provides the best mix of usability and simplicity.