JavaScript object system

Mihai Bazon
www.dynarchlib.com

November 27, 2009

Abstract

This paper describes an elegant "syntax" for object oriented programming in JavaScript, which I developed for DynarchLIB. To understand it, we start with a quick introduction into some of the most confusing JavaScript features—objects, functions and the **this** keyword—with simple code examples. If you are a top JavaScript programmer, this document might provide nothing new (but please read it anyway :-). If you aren't, then this might help you become one.

Contents

1 Advanced JavaScript primer

In case you don't know already, JavaScript is a powerful, dynamic language providing nice OOP features. This section is a quick JavaScript OOP primer. But don't be tricked by the word "quick"—we'll go to discuss some features of JavaScript to advanced level. If you already understand OOP, the meaning of the keyword **this**, nested functions and closures—in Javascript—then you can skip this section.

1.1 Objects

In JavaScript everything is an object. This includes basic types such as numbers or strings. A variable normally contains a *reference* to some object, not a copy of it. Exception to this rule make a few "immutable" types such as numbers, strings or booleans. Here is an example to make this clear:

```
var a = 1;
var b = a;
b = 2;
alert( a === b ); // displays "false" which is correct

var a = [ 1, 2, 3 ];
var b = a;
b[1] = 5;
alert( a == b ); // displays "true", a and b point to the same array
alert( a[1] ); // displays 5; changing b affected a as well
alert( b instanceof Array ); // displays "true", b is an Array object
```

The "===" (respectively "!==") operators evaluate to true only if the operands are (respectively are not) exactly the same object.

There are a few peculiarities related to literal numbers or strings. JavaScript says they are not objects:

```
alert( "foo" instanceof String ); // displays "false"
alert( 5 instanceof Number ); // displays "false"

var a = "foo";
alert( a instanceof String ); // displays "false"

var a = new String("foo");
alert( a instanceof String ); // displays "true"
```

However, we can call methods on a literal string or number directly, which suggests that they really are objects:

```
alert( "foo".toUpperCase() ); // displays "FOO"
alert( (5.127).toFixed(2) ); // displays 5.13
```

If we want to call methods on literal numbers, we have to put them in parentheses for an obvious reason: a number itself can contain the dot character.

What happens in fact is that they aren't objects, but are automatically promoted to real objects when needed.

JavaScript provides a nice literal notation for creating simple objects. They look like hash tables in most other programming languages. An object is in fact some sort of hash table—it maps properties to values. Methods are in fact properties that happen to be functions. The following sample shows two ways to create an object, which are equivalent but one is obviously more verbose:

```
// 1. verbose
var obj = new Object();
obj._foo = "foo";
obj.getFoo = function() {
  return this._foo;
};
alert( obj.getFoo() );

// 2. using literal notation
var obj = {
  _foo : "foo",
  getFoo : function() {
    return this._foo;
  }
};
alert( obj.getFoo() );
```

This may be a good time to observe that functions can be passed around like any other kind of data. They are objects in their own right. This is funny in JavaScript, because as we will see in the next section, objects are, *in a way*, functions. (OK, they're not functions, but they are constructed by functions. So, a function is an object, and an object is made by a function. Sort of a chicken and egg problem.)

Object is a built-in type, the "base class" for all objects—everything automatically inherits from it, even when not specified.

1.2 Classes

JavaScript doesn't have classes *per se*, but functions can act as constructors for objects. Here's how we define a "class":

```
function Foo(prop) {
  this._prop = prop;
};
```

We can call **Foo** a "class" (or a "type", or a "constructor") now, but as you can see, it's really a *function* that initializes an object instance. We can create an object of type **Foo** by calling for example `new Foo(10)`. When a function call is prefixed by the **new** keyword, JavaScript actually creates an object (an instance of that class) and passes (a reference to) it to that function in the **this** keyword. As we will see, the new object "enjoys" all properties defined in the class' (or really, function's) **prototype**.

We can use the **instanceof** operator at "runtime" to find out if an object is an instance of a certain class:

```
var obj = new Foo(10);
alert(obj instanceof Foo); // shows true
```

1.2.1 Adding methods

To add methods to an object, we need to insert them into the infamous **prototype** property (of the "class"):

```
Foo.prototype.getProp = function() {
  return this._prop;
};
```

JavaScript is a truly dynamic language—even if you execute the above code sequentially (note that the getProp method is added *after* the variable **obj** was created), we can now do the following:

```
alert(obj.getProp()); // displays 10
```

What happens in fact is that **obj** doesn't really have a getProp ~~method~~ property. When a property is requested and not found in the object itself, JavaScript searches it in its class' **prototype**. If found there, it is used as if it belonged to the object.

A method added to the **Foo**'s prototype becomes immediately available to all **Foo** instances, *even if they were created before the method was added.* We can extend even core JavaScript objects. For example, here's how we can add to all arrays a method that determines the minimum element:

```
Array.prototype.min = function() {
  var min = this[0], i = this.length;
  while (--i > 0) {
    if (this[i] < min) {
      min = this[i];
    }
  }
  return min;
};

// test it
alert( [ 5, 3, 10, 1, 2 ].min() ); // displays 1
```

In a function assigned to a class' **prototype**, the **this** keyword *typically* refers to an object instance. In the example above, **this** is actually the array `[ 5, 3, 10, 1, 2 ]`. We will discuss more about **this** in section 1.4.

1.2.2 Per-instance methods

I just want to make a point clear: using **prototype** we can add properties or methods that become available to *all* instances of some class. It is possible and very straightforward to add methods only to a particular object instance:

```
obj.getProp2 = function() {
  return 2 * this.getProp();
};
alert( obj.getProp2() ); // displays 20

var newObj = new Foo(15);
alert( newObj.getProp2() ); // ERROR: newObj.getProp2 is not a function
```

1.3 Inheritance

Inheritance is a central point in object oriented programming. In JavaScript, inheritance is set up by simply declaring that a class' prototype is an instance of a certain type. To define a Bar class that inherits from Foo, we need to write the following code:

```
Bar.prototype = new Foo;
function Bar(propForFoo, propForBar) {
  Foo.call(this, propForFoo); // call base class' constructor
  this._barProp = propForBar;
};
```

We can now add new methods in the same way:

```
Bar.prototype.getBarProp = function() {
  return this._barProp;
};
```

Let's test it:

```
var obj = new Bar(10, 20);
alert(obj.getProp()); // this is inherited from Foo; shows 10
alert(obj.getBarProp()); // this comes from Bar; shows 20
alert(obj instanceof Bar); // shows true
alert(obj instanceof Foo); // also shows true: correct inheritance
```

You can notice in the Bar definition that we need to call the base class constructor. One thing, quite annoying, is that the constructor itself needs to know that its "parent" class is Foo. If we later decide that we want to inherit from a different base class, we need to remember to change this in a few places. It would be nice if we could say, for example:

```
this.constructor.base.call(this, propForFoo);
// instead of Foo.call(this, propForFoo);
```

Even though it's more typing, at least we don't have to name the base class explicitly. Turns out it's possible, but at the cost of *even more* typing:

```
// <preamble>
Bar.prototype = new Foo;
Bar.prototype.constructor = Bar;
Bar.base = Foo;
// </preamble>
function Bar(propForFoo, propForBar) {
  this.constructor.base.call(this, propForFoo);
  this._barProp = propForBar;
};
```

There is a lot of boilerplate code in the preamble, but the constructor itself now looks pretty good. It's more generic than if we were to name the base class explicitly.

Adding methods using constructs like **Bar.prototype.method = ...** is pretty bad too. The purpose of this article is to define a way to create objects which is more enjoyable. We'll get there, but first we need to dig through more JavaScript features and pitfalls.

1.3.1 Base class constructors

The following line in the sample above is particularly intriguing:

```
Bar.prototype = new Foo;
```

What happens is that an object of type Foo is actually created right there, in order to declare Bar's parent class. I always found this confusing.

The prototype of the derived class needs to be an object of the base class type[1]. It is used by JavaScript to "provide" methods or properties that are not available in a derived object instance. That's how inheritance works. However, it could be a problem if constructors have any side effects. Here is an example:

[1]We can note though that it doesn't need to be fully initialized, in other words, the constructor doesn't really need to do anything. It's only needed so that properties and methods available in the base class' prototype be copied to the derived class' prototype, and for **instanceof** to work properly.

```
function Foo(arg) {
  alert("Created a new Foo object with arg: " + arg);
}

Bar.prototype = new Foo; // ***
function Bar(arg) {
  Foo.call(this, arg);
}

var b = new Bar("test");
```

This code will display "Created a new Foo object with arg: undefined" right away when it's "parsed", because of the line marked ***. Then it displays the message again, showing "test" instead of "undefined". We can say that the second message is what we intended—we wish it didn't display the first one, but it does because we *have* to call the constructor in order to setup inheritance.

To work around this issue, I designed all my constructors (with only a few exceptions) to do nothing unless they receive at least one argument. The base class constructor would become:

```
function Foo(arg) {
  if (arguments.length) {
    alert("Created a new Foo object with arg: " + arg);
  }
}
```

The above works fine. It only displays the message when we actually instantiate an object—doesn't do anything while the inheritance is setup, because no argument is given on the *** line above. In retrospect, this wasn't such a good idea because there still are cases when I want to create (and initialize) an object that doesn't take any arguments. The following is better:

```
// start by creating a global variable with an unique value
$INHERITANCE = new (function(){});

function Foo(arg) {
  if (arg !== $INHERITANCE) {
    alert("Created a new Foo object with arg: " + arg);
  }
}

Bar.prototype = new Foo($INHERITANCE); // replaces the *** line
// ... the rest of Bar remains the same
```

So the idea is to call the base class constructor with a special, unique argument when setting inheritance. We can then instruct constructors not to do anything at that time.

1.4 The meaning of "this"

In an object method[2], **this** typically refers to an object instance. This keyword is really a hidden function argument—you don't have to declare it in the argument list, and you don't need to send it explicitly when calling a function; JavaScript does that bit of magic for you (and sometimes it's wrong!). But it's really a function argument. This

[2]We will call "object method" a function assigned to a class' prototype, but you have to keep in mind that they are functions and nothing more.

holds true for all OOP languages. What I mean is: the thing that **this** refers to is not decided when you declare a method, but *when you call it.*

JavaScript provides two ways to call a function. One is direct invocation, i.e. func() (the parens after **func** tell JavaScript that we want to call that function right away). Another one is by using the **call** or **apply** methods of the **Function** object. Yes, all functions are in fact objects, and therefore they can and do have methods. Following is a sample to illustrate **this** in a function. We start by defining a plain function that uses the **this** keyword. Note that it's not assigned to any class' prototype—by all means, you should regard it as a function, not as an object method.

```
function getLength() {
  alert( this.length );
};
```

Now let's see various ways to call it:

```
// 1.
getLength(); // this _should_ display "undefined"

// 2.
// as an object method
var obj = { length: 10 };
obj.getLength = getLength;
obj.getLength(); // this displays 10

// 3.
// note that it doesn't have to hold the same name:
var obj = { length: 20 };
obj.gimmeLength = getLength;
obj.gimmeLength(); // displays 20
```

All these are examples where the **this** keyword is implicitly set by JavaScript to a certain meaning. In case 1, which *should* display "undefined", the **this** in getLength is not specified (all browsers will in fact use the **global** (**window**) object).

In the latter cases we call the function by invoking it through a certain object instance (**obj**). In this case, **this** is set to **obj**. It makes no difference that we named it differently in case 3, it still calls the exact same function and passing the correct **this**.

The JavaScript interpreter simply looks at what's before the last dot in expressions like "obj.gimmeLength**()**" and uses it for **this**. Note the parens (), which tell JavaScript that we actually meant to *call* that function. Without the parens, the behavior is completely different. Say, for example, that we continue the code this way:

```
var func = obj.gimmeLength;
func(); // this displays "undefined" or whatever was displayed in case 1.
```

Also, I'm pretty sure that each and every JavaScript programmer hit the following issue:

```
setTimeout(obj.gimmeLength, 15);
```

It displays "undefined", or whatever was displayed in case 1! Why? Because the function is not called right away. obj.gimmeLength is simply a function in no way related to **obj**—it only becomes related *when it is called*, but when setTimeout calls the function after 15 milliseconds, it no longer knows to pass **obj** for **this**.

If we add the parens, something else happens:

```
setTimeout(obj.gimmeLength(), 15);
```

It apparently works, but the function is called *immediately*, and not after 15 milliseconds as we had intended. After 15 milliseconds, we’re likely to see an error in the JavaScript console because we passed an undefined value to setTimeout (the return value from obj.gimmeLength() is **undefined**, since it doesn’t return anything).

So what we need, actually, is a way to define a function that will call our method, in our object instance, no matter what. It’s easy, but not immediately convenient:

```
setTimeout(function(){
  obj.gimmeLength();
}, 15);
```

This passes to setTimeout an “anonymous function” which does the job, because it can access the **obj** variable and call the right method *on it.* Anonymous functions are essential in JavaScript and we will need to talk about them, but for now let’s continue with **this**.

1.4.1 Passing “this” manually

Remember our first sample at inheritance, we called the base class constructor like this:

```
Foo.call(this, propForFoo); // call base class’ constructor
```

and in another sample, which we argued it’s better, we did the following:

```
this.constructor.base.call(this, propForFoo);
```

Could we have written the following instead?

```
this.constructor.base(propForFoo);
^^^^^^^^^^^^^^^^ // this becomes *this*
```

Unfortunately not. What would actually be passed for **this** is “this.constructor”—that’s before the last dot—which isn’t the object instance we need.

For this reason, JavaScript provides two methods in all Function objects: **call** and **apply**. Using these you can explicitly specify what you want the **this** keyword to refer to. In other words, you can *manually* pass the value of **this** for that particular function invocation. Back to our sample above, we can say:

```
var anotherObj = { length: 30 };
getLength.call(anotherObj); // displays 30
```

Both **call** and **apply** are meant to call a function *in the context of some specified object*[3], and pass the rest of the arguments to the function itself. The difference is that **apply** receives the arguments in an array. Here is another sample to illustrate this:

```
function f(a, b, c) {
  alert("I am: " + this + " and my arguments are: " +
        "a: " + a + ", b: " + b + ", c: " + c);
}

f.call("foo", 1, 2, 3);
f.apply("foo", [ 1, 2, 3 ]);
```

Both invocations above display “I am: foo and my arguments are: a: 1, b: 2, c: 3”.

[3]So the **this** for that particular function call points to the given object

1.5 Anonymous/nested functions, closures

JavaScript has "first class" functions. This name is misleading, but what it means really is that you can pass functions around like any other data type. You can send a function as an argument, or return a function from another function. You can nest functions indefinitely.

Variables have either global or function scope[4]—if you prefix a variable definition with the keyword **var**, then it becomes local in the innermost containing function (or global if there's no containing function). Normally, local variables get disposed ("garbage collected") when the function exits, unless there is *another* function that still can access them—such as an inner (nested) function.

Inner functions become "closures" for the context where they are defined. In other words, a nested function has access to variables defined in the outer scope. For as long as a nested function is accessible, those variables are too—the garbage collector cannot dispose them. Closures are one of the most interesting innovation in programming, allowing you to express complicated stuff in a simple manner. No serious JavaScript programming can be possible without understanding closures, so we will devote this section to them.

Here's a basic example:

```
function next(x) {
  return function() {
    return x + 1;
  };
}

var f = next(5);
alert( f() ); // displays 6
```

The function **next** takes one argument (**x**) and returns a function that when called will return the incremented value of **x**. The inner function doesn't take any arguments—it simply returns `x + 1`. It can access the variable **x** from the outer function. With `f = next(5)` we get a "copy" of that nested function which is tied into a particular context where **x** is 5.

Note that *the outer function has already finished execution* by the time we call `f()`. However, `f` can still access the variable **x**. We can continue:

```
var g = next(10);
alert( g() ); // displays 11
alert( f() ); // still displays 6!
```

Calling next(10) *creates a brand new context* with a new **x** variable, and returns a reference to the same nested function but *linked to the new context*. Thus, it doesn't affect `f()` which still displays 6.

So far so good, and I can hear people crying that Java has closures too[5]. However, where it gets interesting is that JavaScript closures can not only access, but also *modify* the variables in their context. For example:

```
function counter(start) {
  return function() {
```

[4]JavaScript 1.7 defines a new "block" scope using the **let** keyword, which is a great addition. Using it you can define variables local to the innermost block, similar to Java or C. However, JavaScript 1.7 is still not widely supported so we will ignore it for now.

[5]http://mihai.bazon.net/blog/the-buzz-of-closures#comment3455

```
    var tmp = start;
    start = start + 1;
    return tmp;
    // or we could have wrote return start++,
    // but I'm striving for clarity here
  }
};

var f = counter(3);
alert( f() ); // displays 3
alert( f() ); // displays 4
alert( f() ); // displays 5

var g = counter(1);
alert( g() ); // displays 1
alert( f() ); // displays 6
alert( g() ); // displays 2
```

With `f = counter(3)` we got a function that, when called, will return consecutive values starting with 3. It returns the next value each time. Note that it's a very natural style of programming—the typical Java or C programmer will create an object, or a structure and manually maintain a property within that object. We don't need to do all this—closures maintain state for us.

1.5.1 Shadowed variables

This is one small point that you want to be aware of. When a variable is declared using **var** in the inner function, or when it's declared in the inner function's argument list, then even if a variable with the same name exists in the outer function it can no longer be accessed. We say that the variable from the inner function "shadows" the one in the upper function. You might or might not want to do this intentionally. Here is an example of a typical error:

```
function foo() {
  var i = 10;
  return function() {
    alert(i);
    var i = 5;
  };
};

var f = foo();
f(); // displays "undefined"!
```

You might think that this displays 10, since `alert(i)` is before `var i = 5`. However, the fact that "`var i`" exists in the inner function makes the outer variable inaccessible. That is why it's a good programming practice (hard to follow, unfortunately) to declare all variables at the start of the function. The standard advice to "declare the variable close to where you use it" from C++/Java languages doesn't apply to JavaScript because whenever it sees a variable declaration, JavaScript makes *all* occurrences of that name through the containing function refer to it. So even if you put "var i" in the middle of the function, JavaScript behaves as if it were at the beginning. The following code *is equivalent*, but this time the reason why it displays "undefined" is obvious:

```
function foo() {
  var i = 10;
  return function() {
    var i;
    alert(i);
    i = 5;
  };
};

var f = foo();
f(); // displays "undefined"!
```

1.5.2 “this” in nested functions

A lot of people find this annoying: in nested functions **this** is not “copied” from the outer scope. In fact, it happens that everything I mentioned in section 1.4 holds true for nested functions as well. It’s that easy. Samples:

```
var obj = {
  _prop = "foo",
  getProp: function() {
    return this._prop;
  },
  getPropFunc: function() {
    return function() {
      return this._prop;
    }
  }
};

alert( obj.getProp() ) // displays "foo"
var f = obj.getPropFunc();
alert ( f() ) // displays "undefined"
alert ( f.call(obj) ) // displays "foo"
```

The inner function returned by getPropFunc() is in no way related to the object, that’s what happens. If we defined it the following way, it would work because the nested function is now a closure and has access to the value that we store in **self**:

```
  getPropFunc: function() {
    var self = this; // store a reference to the object
    return function() {
      return self._prop; // avoid "this" here, use self instead
    }
  }

  var f = obj.getPropFunc();
  alert( f() ); // displays "foo"
```

1.5.3 Performance of closures

Closures are slower than top-level functions. This happens because they carry the bound variables from their enclosing context—otherwise said, the interpreter needs to change context whenever it calls an inner function, which wouldn’t happen if all functions were defined in the global scope. *But*, in my experience (and I’m talking tens of thousands of lines, JavaScript code which would have been *a lot* bigger and uglier without closures)—they’re *fast enough*. My first rule of thumb is “don’t optimize *unless you have to*”.

One example where speed is critical are chess engines. See for instance my

> father's JavaScript chess engine[6]. He implemented it largely with global variables and functions. We agreed it's ugly so, at some point, we tried to put everything inside an outer function, and only export a few entry points—thus emulating an object. Since the variables were now local, we can put more than one game in the same page, with no major changes in the code.
>
> It was almost two times slower than with globals (1800 nodes per second versus 3200 nodes per second).
>
> A chess engine, even an average one, needs to analyze some thousands chess positions per second in order to provide a reasonable response.
>
> I happen to know a bit of computer chess theory, so let me tell you what "1800 nodes per second" means: it's the number of *final positions that have been evaluated*. Imagine a tree that has 1800 leaf nodes. The *total* number of nodes is a lot larger, if we count the intermediary ones. 1800 evaluated nodes means that the program has in fact generated hundreds of thousands of moves and probably tens of thousands intermediate positions, before calling the evaluate function. In *one second*.
>
> That's what I mean when I say that closures are fast enough. They don't work well for a chess engine, but the speed impact on most code that we write, for most day to day problems, goes unnoticeable, while the benefits in code size, clarity and development time are enormous.

Technically, that's all there is to know about inner functions/closures. It's up to your imagination now to create wonderful things with them. One of these wonderful things follows in the next section.

1.6 Making sure we pass the right "this"

Using closures, we can now easily create a function that calls an object method passing the right value for **this**, no matter how it's called. So we can solve the `setTimeout` problem once and for all. Here is the code of a function named "curry"[7], rather verbose for clarity:

```
function curry(object, method) {
  var args = [], i = 2;
  while (i < arguments.length) {
    args.push(arguments[i++]);
  }
  return function() {
    var newArgs = [], i = 0;
    while (i < arguments.length) {
      newArgs.push(arguments[i++]);
    }
    return method.apply(object, args.concat(newArgs));
  };
};
```

This is one of the most useful functions I wrote. I use it so frequently that I gave it a single-character name in DynarchLIB. Here's how to use it:

```
var obj = {
  length: 10,
  getLength: function() {
```

[6] http://docere.ro/momente/doChess

[7] See http://en.wikipedia.org/wiki/Currying for details. In DynarchLIB this function is called "closure", but I now think that "curry" is a better name.

```
    alert(this.length);
  }
};
obj.getLength(); // works
setTimeout(obj.getLength, 15); // remember, this doesn't work
setTimeout(curry(obj, obj.getLength), 15); // this works!
```

What does it do? **curry** returns a function that will call your object method with the correct **this**. The inner function has access to the variables **object** and **method** from the outer scope, and can thus do the right thing even when called with no arguments. There is *no way* to do this without closures.

You can notice that it does something with an invisible **arguments** variable. This variable is an implicit local variable (standard in JavaScript) which contains all the arguments that were passed to the function. Using it, **curry** saves any extra arguments given (after **method**) and passes them to the object method when it will be called, adding any arguments passed at invocation time. An example should clear any confusion:

```
function hello(a, b, c) {
  alert("I am " + this + " and got: " +
        [a, b, c].join(", "));
};

var f = curry("foo", hello, 1, 2, 3);
f(); // displays "I am foo and got: 1, 2, 3"

var g = curry("bar", hello, 4);
g(5, 6); // displays "I am bar and got: 4, 5, 6"
```

In the example above, we transformed a function "hello" which takes 3 arguments and expects **this** to be a string, into other functions **f** and **g** which are "partial applications" of **hello**. **f** actually memorized all the required arguments, so we can call it with no arguments and it will correctly invoke **hello**. On the other hand, **g** only contains the object and one argument (4), allowing for the other two to be passed at invocation time.

1.6.1 Extending Function objects

As I mentioned, the **curry** function is so important and will end up being so frequently used, that it would make sense if it were a Function method, at the same level as **call** and **apply**. Here's how we can do that:

```
Function.prototype.curry = function(object) {
  var method = this, copy = Array.prototype.slice, args = copy.call(arguments, 1);
  return function() {
    var newArgs = copy.call(arguments);
    return method.apply(object == null ? this : object, args.concat(newArgs));
  };
};

// and let's create a shorter alias:
Function.prototype.$ = Function.prototype.curry;
```

Note that I wrote it simpler by using Array.prototype.slice to copy arguments. Also, in the special case where you pass **null** for object, it will actually pass the value of **this** at invocation time.

After running this code, we can rewrite the previous samples like this:

```
setTimeout(obj.getLength.curry(obj), 15);
var f = hello.curry("foo", 1, 2, 3);
var g = hello.curry("bar", 4);

// or, using the alias:
setTimeout(obj.getLength.$(obj), 15);
var f = hello.$("foo", 1, 2, 3);
var g = hello.$("bar", 4);
```

2 JavaScript macros

"Experienced Lisp programmers divide up their programs differently. [...]. They follow a principle which could be called bottom-up design—changing the language to suit the problem. In Lisp, you don't just write your program down toward the language, you also build the language up toward your program. [...]. Like the border between two warring states, the boundary between language and program is drawn and redrawn, until eventually it comes to rest along the mountains and rivers, the natural frontiers of your problem. In the end your program will look as if the language had been designed for it. And when language and program fit one another well, you end up with code which is clear, small, and efficient."

Paul Graham in "Programming Bottom-Up"[8]

I'm kidding, alright? It would be too cool if JavaScript had real macros. Maybe I'll live up to the day when it will, but until then we can still do something about the long boilerplate code that we need to write in order to define objects. It's true that Paul Graham talks about Lisp in the above quote, but JavaScript is closer to Lisp than many people think.

We need to observe that JavaScript code is being executed as the browser[9] reads the file. There *is* an invisible compilation step, which makes functions available even if they are defined later in the code, but disregarding this, the browser *starts executing the code immediately.* I will call *this* moment "compile time" and it has nothing to do with the invisible compilation step that I mentioned. During this period we can automate tasks to do the boring parts for us. After all files have been read and executed, we enter "runtime"—the part that's actually going to make use of our great code.

Of course, I know that other JavaScript frameworks have solved the boilerplate problem. I'm just proposing a different solution. Instead of using a "declarative" syntax (passing a long hash table to some "Class.create" function), I prefer the "executive way", so to call it—code that *runs*, instead of declarations.

2.1 The DEFCLASS "macro"

Before diving into how I did it, here is first how I use it. The following code shows how I define a "class" in DynarchLIB:

```
DEFCLASS("MyObject", null, function(D, P){
```

[8] http://www.paulgraham.com/progbot.html

[9] I talk about a "browser" because this is the most common platform. But JavaScript becomes a general programming language in its own right, and any other platform would do the same.

```
  D.DEFAULT_ARGS = {
    _something: [ "something", "Default Value" ]
  };

  D.CONSTRUCT = function() {
    alert("Constructing new object of type: " + this._objectType);
  };

  var privateVar = 123;

  D.getPrivateVar = function() {
    return privateVar;
  };

  P.getSomething = function() {
    return this._something;
  };

});

var obj = new MyObject({ something: "foo" });
alert( obj.getSomething() ); // displays "foo"

var otherObj = new MyObject({});
alert( otherObj.getSomething() ); // displays "Default Value"

alert( MyObject.getPrivateVar() ); // displays 123
```

We can already see some magic provided by DEFCLASS[10]:

- it immediately calls a function that we supplied passing two arguments which I'm going to call **D** and **P** for historical reasons. **D** is a reference to the class (i.e. **MyObject**) and **P** is the class' prototype (**MyObject.prototype**). This function will add methods and properties to our object and can conveniently store any private data in local variables.
- it implements a generic way to pass arguments to constructors, via the DEFAULT_-ARGS property of the *class*. The keys in DEFAULT_ARGS become properties of the object instance. The values are arrays with two elements, the first one being the name of the property passed to the constructor, and the second is the default value if none was supplied.
- it provides an _objectType property in all objects. This property contains the name of the object class that was instantiated, which is otherwise quite difficult to retrieve in JavaScript.
- finally, no more boilerplate! :-)

The second argument to DEFCLASS (which is **null** in the example above) is a reference to the base class. Inheritance made easy!

```
DEFCLASS("MyDerivedClass", MyObject, function(D, P){

  D.DEFAULT_ARGS = {
    _anotherThing: [ "anotherThing", null ]
  };

  P.getAnotherThing() {
```

[10]In DynarchLIB this is named DEFINE_CLASS. I now think that DEFCLASS is a nicer name, in spirit of Lisp

```
    return this._anotherThing;
  };

});

var obj = new MyDerivedClass({ something: 1, anotherThing: 2 });
alert( obj.getSomething() ); // displays 1
alert( obj.getAnotherThing() ); // displays 2
alert( obj instanceof MyDerivedClass ); // true
alert( obj instanceof MyObject ); // true
```

As you can see, the arguments of the base class (defined in MyObject.DEFAULT_-ARGS) automatically become available in the derived class too. We don't have to write a constructor if we don't need one. We don't even have to call the base class constructor—this is taken care of by DEFCLASS. I never found situations where I needed to call the base class constructor manually, but I did at times want to catch and modify some arguments at construction time. DEFCLASS supports a few hooks that you can add in order to customize its behavior for a certain object.

2.2 A more complex example

What follows is a simplified version of real code that exists in DynarchLIB. We start by creating a base class called DlEventProxy—it provides an infrastructure for event handlers within an object.

```
DEFCLASS("DlEventProxy", null, function(D, P){

  D.CONSTRUCT = function() {
    this.__eventHooks = {};
    this.registerEvents(this.DEFAULT_EVENTS);
  };

  P.DEFAULT_EVENTS = [ "onDestroy" ];

  P.FINISH_OBJECT_DEF = function() {
    var moreEvents = this.constructor.DEFAULT_EVENTS;
    if (moreEvents)
      this.DEFAULT_EVENTS = this.DEFAULT_EVENTS.concat(moreEvents);
  };

  P.registerEvents = function(evs) {
    var h = this.__eventHooks, i = 0, e;
    while ((e = evs[i++])) {
      e = e.toLowerCase();
      if (!h[e])
        h[e] = [];
    }
  };

  P.addEventListener = function(ev, handler) {
    getHooks.call(this, ev).push(handler);
  };

  P.removeEventListener = function(ev, handler) {
    getHooks.call(this, ev).remove(handler);
  };

  P.callHooks = function(ev) {
    return this.applyHooks(ev, arguments);
  };

  P.applyHooks = function(ev, args) {
    getHooks.call(this, ev).foreach(function(f){
      f.apply(this, args);
    }.$(this));
  };

  // this is a private function, we call it using .call or .apply
  function getHooks(event) {
    return this.__eventHooks[event.toLowerCase()];
  };

});
```

The code in DynarchLIB is actually a bit more complicated, but the above captures the essence. Almost all other objects inherit from DlEventProxy in order to provide events. Various objects will have different events. For example, all widgets define an “onClick” event which is triggered by the framework upon a mouse click on the widget. On the other hand, an object that fetches data from the server might want to define an “onDataReady” event.

Prior to DEFCLASS, all objects needed to call this.registerEvents(array of events) somewhere in constructor, *after* the constructor of the base class (thus including DlEventProxy) was called, *but before* a listener was added for that event. It was pretty messy to keep track of this code, which is why I added a FINISH_OBJECT_DEF hook in DEFCLASS. What it does—if this function is defined, it is called automatically by DEFCLASS on the new class’ prototype (so **this** in that function is not a real object instance, but the **prototype** of its class). Since the prototype is inherited, what this amounts to is that whenever we define a DlEventProxy subclass, the function above gets called and collects any events defined in constructor’s DEFAULT_EVENTS property into the new object prototype. The single registerEvents call above will then register all events in one shot, at the right time. Let’s see how a subclass could look like:

```
DEFCLASS("DlWidget", DlEventProxy, function(D, P){
  D.DEFAULT_EVENTS = [ "onMouseDown", "onMouseUp", "onClick", ... ];
```

That’s *all* it needs to do, really, for defining events. If we want, we can add event listeners in the constructor directly now, without having to call registerEvents first:

```
  D.CONSTRUCT = function() {
    this.addEventListener("onClick", function() {
      alert("You clicked me!");
    });
  };
```

But let’s now show more power of FINISH_OBJECT_DEF:

```
  P.classNames = [];
  P.FINISH_OBJECT_DEF = function() {
    D.BASE.FINISH_OBJECT_DEF.apply(this, arguments);
    this.classNames = this.classNames.concat([ this._objectType ]);
  };
```

D.BASE is provided by DEFCLASS and points to the base class’ **prototype**, in this case DlEventProxy.prototype. We can conveniently use this shortcut to call base class methods.

We now defined a new FINISH_OBJECT_DEF function which (1) takes care to call the base class’ method first and (2) maintains an array of class names in the object’s prototype (remember that _objectType is provided by DEFCLASS). All objects now inherited from DlWidget (and DlWidget itself) will provide a classNames property that follows the inheritance graph. For example, DlButton inherits from DlAbstractButton which inherits from DlWidget—so a DlButton's classNames property will be the following:

```
[ "DlWidget", "DlAbstractButton", "DlButton" ]
```

All this happens automatically when you declare a derived class. Note that we can't simply use **push** above—if we did, we would modify intermediate class' properties as well—we really need to copy the whole array and add the new object type. But this array will be *shared* across all object instances of, say, DlButton.

You don't need to worry about performance impact—there is *no* performance impact. The FINISH_OBJECT_DEF functions are executed at "compile time", if I may so—they run as classes are declared, but never when an object is instantiated.

2.3 Implementation of DEFCLASS

In this section I will write the code that implements the DEFCLASS "macro". I won't be copy/pasting from DynarchLIB because there it's a little more hairy. I'm using all-caps names for global functions and variables.

```
// we described the importance of this variable in "base class constructors"
$INHERITANCE = new (function(){});

// helper function that copies arguments into an object instance,
// based on the class' DEFAULT_ARGS property
function SET_DEFAULTS(targetObj, defaultArgs, explicitArgs) {
  if (!explicitArgs) {
    explicitArgs = {};
  }
  for (var targetPropName in defaultArgs) {
    var propDef = defaultArgs[targetPropName];
    var explicitPropName = propDef[0];
    var value = propDef[1];
    if (explicitPropName in explicitArgs) {
      value = explicitArgs[explicitPropName];
    }
    targetObj[targetPropName] = value;
  }
}

// and the real thing
function DEFCLASS(name, base, definition, hidden) {
  // because D is a function (see below) we can access it
  // even before it's declared, thanks to JS compilation step
  if (name) {
    D.name = name;
  }
  // allow one to send the name of the base class, instead of constructor
  if (typeof base == "string") {
    base = window[base];
  }
  // setup inheritance
  if (base) {
    D.prototype = new base($INHERITANCE);
    D.BASE = base.prototype;
  }
  // define the constructor: the following function will construct
  // an instance of our object. It's not called *now*.
  function D(args) {
    if (args !== $INHERITANCE) {
      // allow definition to "fix" some of the arguments
      if (D.FIXARGS instanceof Function) {
        D.FIXARGS.apply(this, arguments);
      }
      // use the default args to initialize object properties
      if (D.DEFAULT_ARGS) {
        SET_DEFAULTS(this, D.DEFAULT_ARGS, args);
      }
      // allow definition to hook in before we call base class constructor
      if (D.BEFORE_BASE instanceof Function) {
        D.BEFORE_BASE.apply(this, arguments);
      }
      // call base class' constructor
      if (base) {
        base.apply(this, arguments);
      }
      // allow definition to hook in at construction time
      if (D.CONSTRUCT instanceof Function) {
        D.CONSTRUCT.apply(this, arguments);
      }
    }
  };
  // export the new class if it has a name and it's not requested hidden
  if (name && !hidden) {
    window[name] = D;
  } else {
    D.HIDDEN = true;
  }
  var P = D.prototype;
  P.constructor = D;
  P._objectType = name;
  // finally, call the definition *right away*!
  if (definition) {
    definition(D, P);
  }
  // if definition added a FINISH_OBJECT_DEF function in the object's prototype
  // (or if maybe it came from a base class) then call it immediately!
  if (P.FINISH_OBJECT_DEF instanceof Function) {
    P.FINISH_OBJECT_DEF();
  }
  return D;
}
```

It might seem a little twisted to follow the logic of DEFCLASS. Again, let's discuss the arguments that it receives:

1. **name** — this is a string and it's the name of the newly defined class. If you don't want to export the new class, you can pass **null**. Sounds pointless, but there are situations where you want just that.
2. **base** — this is a reference to the base class constructor, or a string containing the name of the base class. If the new class doesn't inherit from a base class, you can pass **null** here.
3. **definition** — this is a function that builds the new class. If given, it is called by DEFCLASS immediately with two arguments: a reference to the constructor (**D**), and a reference to its prototype (**P**). In order to define object methods, **definition** can simply add them to **P**. To define "class" ("static") methods, they can be added to **D**.

definition can also add some special methods that will affect the way construction works (see "hooks" in 2.3.1 below).

4. **hidden** — this is an optional boolean value that specifies if the newly defined class is "hidden". If you pass **true** here, the new class is not "exported". Sounds similar to the first argument, but sometimes you may want to create a class that has a name yet it's not exported.

DEFCLASS defines an "universal constructor"—all objects will actually be constructed by the same function, which is the inner function **D** in the code. Wait, not *exactly* the same, as it will be linked to a different context for each object, but here goes:

```
DEFCLASS("Foo");
DEFCLASS("Bar", Foo);

alert(Foo);
alert(Bar);
```

Both the alerts above will display:

```
function D(args) {
    if (args !== $INHERITANCE) {
        if (D.FIXARGS instanceof Function) {
            ...
```

In practice, I never needed to "stringify" a function so I don't mind that. However, it could confuse some debuggers or error messages. That's life. The **D** function itself is "exported" with a different name (the name we give as the first argument to DEFCLASS), but there's no way (well, none that I could figure out) to let JavaScript know this, except for using **eval** but this I'm trying to avoid.

2.3.1 Hooks supported by DEFCLASS

The **definition** function that we supplied can add certain functions that will get called by the "universal constructor" at various stages during object initialization. All these constructor hooks are called in the context of the newly created object (so **this** will refer to the new object instance) and will receive all arguments passed at construction time.

FIXARGS this is a function intended to "fix" the arguments at construction time. It is called before SET_DEFAULTS, so it has a chance to add/remove/modify the arguments that were actually passed to the constructor. It's intended to work with a single hash argument.

BEFORE_BASE is a function that gets called before the constructor of the base class was invoked.

CONSTRUCT is a function that is called after the base class constructor was invoked. The object is now fully initialized, contains all the properties specified in DEFAULT_ARGS and this function can do further customization, including calling any methods on the object.

One special "prototype hook" is defined: **FINISH_OBJECT_DEF** which is called immediately after **definition** finished execution. Since it's not called on an object instance, the **this** in that function refers to the class' prototype. As you could saw in examples in the previous section, this function can do a lot of automated things on derived objects.

2.3.2 Properties added by DEFCLASS

DEFCLASS also adds a few properties to the prototype or the constructor of the newly defined class (named MyClass below):

- `MyClass.name == "MyClass"`
- `MyClass.HIDDEN == true`, if the new class is not exported
- `MyClass.BASE == BaseClass.prototype`, if MyClass inherits from BaseClass
- `MyClass.prototype.constructor == MyClass` (this way you can access the constructor in an object instance using **this.constructor**)
- `MyClass.prototype._objectType == "MyClass"`

2.4 More "macros" on top of DEFCLASS

I don't know if I convinced you or not, but to me DEFCLASS is already the best way to define objects in JavaScript. Prior to this, I used some arcane constructs—which still automated tasks, but required me to write a constructor even in cases where I didn't need one, call base constructors and export the new class manually (which was using "eval", again, in order to automate some tasks). The released DynarchLIB still contains this awful code. After switching to DEFCLASS things were cleaner and the minified[11] code got almost 40K smaller. It's a significant gain.

However, DEFCLASS allows us to further minimize the code on particular cases. This section shows some.

2.4.1 Hidden classes

As I mentioned, DEFCLASS can easily be made to avoid exporting the new class if you pass **true** for the last argument. This is not nice though, because the **definition** function will be fairly large so you can't see just taking a glance at the code if a certain class is hidden or not.

The DEF_HIDDEN_CLASS "macro" helps clarity in this case:

```
function DEF_HIDDEN_CLASS(name, base, definition) {
  return DEFCLASS(name, base, definition, true);
}

// Now you can call it like this:
var Foo = DEF_HIDDEN_CLASS("Foo", Bar, function(D, P){
  // ... define the object here
};
```

Hidden classes are useful. For instance, many widgets in DynarchLIB support some custom way to do drag'n'drop. There is a DlDrag base class that provides a lot of infrastructure needed—such as noticing when a d'n'd operation starts, noticing the object below the mouse as it is dragged, or deciding what to do when the widget is dropped. The DlDrag class needs to be subclassed to provide useful stuff for particular widget types, but in the vast majority of cases there was no need to export the derived class since no other part of the code needs it. Here's how it's done:

[11] With no unnecessary whitespace, no comments and private variables renamed to single-letters. The YUI compressor can do this (http://developer.yahoo.com/yui/compressor/)

```
DEFCLASS(MyWidget, "DlWidget", function(D, P){

  D.FIXARGS = function(args) {
    args.drag = new MyWidgetDrag();
  };

  var MyWidgetDrag = DEF_HIDDEN_CLASS("MyWidgetDrag", DlDrag, function(D, P){
    // ... definition of drag class here
  });

});
```

In the above code, MyWidgetDrag is a local variable in the anonymous function (**definition**) that we pass to DEFCLASS. It is a constructor for “MyWidgetDrag” objects, but it’s “private”—you can’t access it from anywhere in the code.

2.4.2 Singleton objects

The “singleton” paradigm is another good example where hidden classes are useful. A singleton is a class that can only be instantiated once, or never, during the execution of a program. Instantiation is “lazy”—it happens only when needed, if ever. Of course, you can easily do this with standard JavaScript constructs:

```
function Foo() { ... constructor here ...}

function getFoo() {
  if (!Foo.INSTANCE) {
    Foo.INSTANCE = new Foo();
  }
  return Foo.INSTANCE;
};
```

The function getFoo() creates the Foo instance and returns it. On subsequent calls, it will return the existing instance instead of creating a new one. This does the job if nobody else is using the code, but the Foo constructor is public. When someone else uses the code, perhaps someone who didn’t know that Foo isn’t meant to be instantiated more than once, she will be tempted to call “new Foo” again instead of using the getFoo() method. The key to define a singleton class is to hide the constructor.

In C++ you would do this by declaring the constructor “private”, and provide a static function within the object (thus it will have access to the constructor) that would maintain the “INSTANCE” pointer. This can be factored into generic code using C++ templates.

In JavaScript we can do the following:

```
function DEF_SINGLETON(name, base, definition) {
  var constructor = DEF_HIDDEN_CLASS(name, base, definition), instance = null;
  function getInstance() {
    return instance || ( instance = new constructor() );
  };
  if (name) window[name] = getInstance;
  return getInstance;
};
```

The above function receives the same arguments as DEF_HIDDEN_CLASS and it defines a global function that returns (creates if it wasn’t already) the instance of our singleton.

The singleton constructor function is private—a third party programmer using our code can't accidentally instantiate a new object of that type. Sample usage:

```
DEF_SINGLETON("DlSystem", DlEventProxy, function(D, P){
  D.DEFAULT_EVENTS = [ "on-rpc-start", "on-rpc-stop", ... ];
});

// then here's how we can use it:
DlSystem().callHooks("on-rpc-start");
```

As you can see, it creates a global function with the given name. This function should not be called with **new**—instead, calling it directly will return the instance of our singleton (creating it on the first call). You can also create a "hidden singleton" if you pass **null** for **name**. The "getInstance" function is returned and you can use it in your private context.

3 Epilogue

> Prince Wang's programmer was coding software. His fingers danced upon the keyboard. The program compiled without an error message, and the program ran like a gentle wind.
>
> *"Excellent!"* the Prince exclaimed, *"Your technique is faultless!"*
>
> *"Technique?"* said the programmer turning from his terminal, *"What I follow is Tao - beyond all techniques! When I first began to program I would see before me the whole problem in one mass. After three years I no longer saw this mass. Instead, I used subroutines. But now I see nothing. My whole being exists in a formless void. My senses are idle. My spirit, free to work without plan, follows its own instinct. In short, my program writes itself. True, sometimes there are difficult problems. I see them coming, I slow down, I watch silently. Then I change a single line of code and the difficulties vanish like puffs of idle smoke. I then compile the program. I sit still and let the joy of the work fill my being. I close my eyes for a moment and then log off."*
>
> Prince Wang said, *"Would that all of my programmers were as wise!"*
>
> Geoffrey James, "The Tao of Programming" [4.4]

The "master programmer" says *"my program writes itself"*—isn't that beautiful? This article showed how you can do this, to some extent, in JavaScript. If it had real macros, we could push this even further by adding new syntax to the language. I don't know if a proper macro system will ever exist for JavaScript—in fact, of all languages, only Lisp has a proper macro system, which is only made possible by its weird syntax. As Paul Graham noted, "if you add that final increment of power, you can no longer claim to have invented a new language, but only to have designed a new dialect of Lisp"[12].

Feedback appreciated

I welcome any feedback on this article—please send your comments and suggestions to mihai@bazon.net, or mihai.bazon@gmail.com (I tend to read GMail less frequently).

[12] http://www.paulgraham.com/diff.html