Object Oriented Programming in JavaScript

by Mike Koss
March 26th, 2003

Introduction

Most writers of JavaScript use it as a simple scripting engine to create dynamic web pages. And web designers have come to use the object-model that is defined in Internet Explorer to enable them to manipulate the contents of a web page. But most developers do not realize that JavaScript has a powerful object oriented capability in it's own right. While not strongly typed, this interpreted language can support sophisticated object-oriented paradigms including:

- Encapsulation
- Polymorphism
- Inheritance

Through a series of examples (which, for the curious reader, are actually snippets of live JavaScript code embedded within this page), I will demonstrate how objects can be used in JavaScript and how these object oriented paradigms can be best implemented.

Simple Objects

The simplest object oriented construct in JavaScript is it's built in *Object* data type. In JavaScript, objects are implemented as a collection of named properties. Being an interpreted language, JavaScript allows for the creation of any number of properties in an object at any time (unlike C++, properties can be added to an object at any time; they do not have to be pre-defined in an object declaration or constructor).

So, for example, we can create a new object and add several add-hoc properties to it with the following code:

```
obj = new Object;
obj.x = 1;
obj.y = 2;
```

Which creates a JavaScript object which I will represent graphically like this:

obj	
x	1
y	2
prototype properties	
constructor	function Object

Note that in addition to the *x* and *y* properties that we created, our object has an

additional property called *constructor* that points (in this case) to an internal JavaScript function.

Object Constructors

JavaScript further allows for types to be defined by defining our own constructors for an object type:

```
function Foo()
{
    this.x = 1;
    this.y = 2;
}

obj1 = new Foo;
```

obj1	
x	1
y	2
prototype properties	
constructor	function Foo

Note that we can now create as many *Foo* type objects as we want, all of whom will be properly initialized to have their *x* and *y* properties set to 1 and 2, respectively.

A Simple Method Implementation

In order to *encapsulate* the functionality of an object, and hide the implementation from the caller, we need to be able to create methods on an object. JavaScript allows you to assign any function to a property of an object. When you call that function using *obj.Function* syntax, it will execute the function with *this* defined as a reference to the object (just as it was in the constructor).

```
function Foo()
{
    this.x = 1;
    this.y = 2;
    this.Bar = MyMethod;
}

function MyMethod(z)
{
    this.x += z;
}

obj2 = new Foo;
```

obj2	
x	1
y	2
	function MyMethod

Bar	
prototype properties	
constructor	function Foo

Now we can simply call the Bar function as a method of the object:

```
obj2.Bar(3);
```

obj2	
x	4
y	2
Bar	function MyMethod
prototype properties	
constructor	function Foo

So, you can see how Encapsulation can be accomplished by defining the constructor and defining the methods on an object that will hide the implementation details from the caller. Also, because JavaScript is not strongly typed, we can achieve polymorphism simply by defining objects that have the same named methods.

```
function Foo()
{
    this.x = 1;
    this.DoIt = FooMethod;
}

function FooMethod()
{
    this.x++;
}

function Bar()
{
    this.z = 'Hello';
    this.DoIt = BarMethod;
}

function BarMethod()
{
    this.z += this.z;
}

obj1 = new Foo;
obj2 = new Bar;
```

obj1	
x	1
DoIt	function FooMethod
prototype properties	

constructor	function Foo

obj2	
z	Hello
DoIt	function BarMethod
prototype properties	
constructor	function Bar

```
function Poly(obj)
{
    obj.DoIt();
}

Poly(obj1);
Poly(obj2);
```

obj1	
x	2
DoIt	function FooMethod
prototype properties	
constructor	function Foo

obj2	
z	HelloHello
DoIt	function BarMethod
prototype properties	
constructor	function Bar

Using Prototypes to Implement Methods

It is awkward to have to create a dummy name for each method function, and then have to assign the method property in the constructor to each method function. I have found that there is a more direct solution, which relies on JavaScript's *prototype* mechanism.

Each object can reference a *prototype* object that can contain it's own properties. It is used as a kind of *back-up* object definition. When code references a property that does not exist in the object itself, JavaScript will automatically looks in the prototype for a definition. And in fact, a prototype can refer to another prototype in a chain leading up to the prototype association with the base Object constructor. This sort of template model, can be used to simplify our method definitions, as well as form the basis for a powerful inheritance mechanism.

Prototype's in JavaScript are associated with an object's constructor. So, in order to assign a prototype to an object, if must first be assigned to the constructor's *prototype* member. Then, when an object is constructed from that constructor function, the object will refer to the constructor's prototype.

```
function Foo()
{
    this.x = 1;
}

Foo.prototype.y = 2;
obj = new Foo;
document.write('obj.y = ' + obj.y);
```

obj.y = 2

obj	
x	1
prototype properties	
constructor	function Foo **prototype** **y** 2
y	2

The obj object appears to have a y property, even though we never assigned one to obj explicitly. When we make a reference to obj.y, JavaScript will instead return the value of obj.constructor.prototype.y. We can confirm this behavior by changing the value of the prototype, and inspecting the (apparent) values of the object:

```
Foo.prototype.y = 3;
document.write('obj.y = ' + obj.y);
```

obj.y = 3

obj	
x	1
prototype properties	
constructor	function Foo **prototype** **y** 3
y	3

We can also see, that once we have initialized a *private* value for a property, the value stored in the prototype no longer takes affect:

```
obj.y = 4;
Foo.prototype.y = 3;
```

obj	
x	1
y	4

prototype properties	
constructor	function Foo **prototype** **y** 3

Prototype Method Naming

I found a clever way to then define methods of the class prototype directly, without having to individually name the function:

```
function Foo()
{
    this.x = 1;
}

function Foo.prototype.DoIt()
{
    this.x++;
}
obj = new Foo;
obj.DoIt();
```

obj	
x	2
prototype properties	
constructor	function Foo **prototype** **DoIt** function Foo.prototype.DoIt
DoIt	function Foo.prototype.DoIt

Prototype-based Subclassing

One can create a prototype chain, and use that for a type of subclassing of objects. Note that in this method, we create an instance of the base-class object, and assign it to be our class constructor's prototype object. By doing so, all of the objects that we create, will inherit all the members (and methods) of the base class object. But note that the base classes constructor is called *only once*. This is unlike C++ where the base class's constructor is called for each creating of the derived class. I'll show an alternate method of building a class heirarchy later on if that behavior is required.

```
function TextObject(st)
{
    this.st = st;
    this.fVisible = true;
}

function TextObject.prototype.Write()
{
    document.write('
```

```
' + this.st);
}

function ItalicTextObject(st)
{
    this.st = st;
}

ItalicTextObject.prototype = new TextObject('x');

ItalicTextObject.prototype.Write = ITOWrite;
function ITOWrite()
{
    document.write('
' + this.st + '');
}

obj1 = new TextObject('Hello, mom');
obj2 = new ItalicTextObject('Hello, world');
obj1.Write();
obj2.Write();
```

Hello, mom
Hello, world

obj1	
st	Hello, mom
fVisible	true
prototype properties	
constructor	function TextObject **prototype** **Write** function TextObject.prototype.Write
Write	function TextObject.prototype.Write

obj2	
st	Hello, world
prototype properties	
constructor	function TextObject **prototype** **Write** function TextObject.prototype.Write
fVisible	true
Write	function ITOWrite

There are a couple of problems with this system. First, the constructor of the base class is not called when constructing each derived class. If the constructor does anything more than initializing member variables to some static values, this may not be very desirable. Second, note that I was unable to use the "function Obj.prototype.Method" paradigm for the definition of derived class members. This is because I need to add

these method definition to the newly created prototype object, rather than add them to the default prototype for the derived constructor function.

An Alternate Subclassing Paradigm

I've come up with a paradigm which more closely mirrors the notion of class and subclass definitions of C++ and addresses some of the drawbacks of the pure prototype chain style of subclassing. It requires a new method DeriveFrom be added to the Function class.

```
function Function.prototype.DeriveFrom(fnBase)
{
    var prop;

    if (this == fnBase)
        {
        alert("Error - cannot derive from self");
        return;
        }

    for (prop in fnBase.prototype)
        {
        if (typeof(fnBase.prototype[prop]) == "function" && !this.prototype[prop]
            {
            this.prototype[prop] = fnBase.prototype[prop];
            }
        }

    this.prototype[fnBase.StName()] = fnBase;
}

function Function.prototype.StName()
{
    var st;

    st = this.toString();
    st = st.substring(st.indexOf(" ")+1, st.indexOf("("))

    return st;
}

function TextObject(st)
{
    this.st = st;
    this.fVisible = true;
}

function TextObject.prototype.Write()
{
    document.write('
' + this.st);
}

function TextObject.prototype.IsVisible()
{
    return this.fVisible;
}

function ItalicTextObject(st)
{
```

```
    this.TextObject(st);
}

ItalicTextObject.DeriveFrom(TextObject);

function ItalicTextObject.prototype.Write()
{
    document.write('
' + this.st + '');
}

obj1 = new TextObject('Hello, mom');
obj2 = new ItalicTextObject('Hello, world');
obj1.Write();
obj2.Write();
```

Hello, mom
Hello, world

obj1	
st	Hello, mom
fVisible	true
prototype properties	
constructor	function TextObject **prototype** **Write** function TextObject.prototype.Write **IsVisible** function TextObject.prototype.IsVisible
IsVisible	function TextObject.prototype.IsVisible
Write	function TextObject.prototype.Write

obj2	
st	Hello, world
fVisible	true
prototype properties	
constructor	function ItalicTextObject **prototype** **Write** function ItalicTextObject.prototype.Write **IsVisible** function TextObject.prototype.IsVisible **TextObject** function TextObject **prototype** **Write** function TextObject.prototype.Write **IsVisible** function TextObject.prototype.IsVisible
IsVisible	function TextObject.prototype.IsVisible
	function TextObject

TextObject	prototype	
	Write	function TextObject.prototype.Write
	IsVisible	function TextObject.prototype.IsVisible
Write	function ItalicTextObject.prototype.Write	

We have an added benefit that we can derive from more than one base class (multiple inheritance).

References

surprise! JavaScript is object-oriented - October 1997 article on Builder.Com written by Dan Shafer. Explores and explains some of these issues (see also his follow up article in November 1997).

JavaScript reference - MSDN reference. See especially definitions of:
constructor
prototype
hasOwnProperty
isPrototypeOf
propertyIsEnumerable

watch() method documentation - DevGuru. Allows a function to be called whenever an object property is changed, is not supported by Internet Explorer's implementation of JScript.

Document Object Model for Internet Explorer - MSDN web site.

Checking for a Prototype Chain - Person/Employee example of subclassing from this page by Yehuda Shirna (Doc JavaScript)

Prototype Property - Wrox JavaScript Programmers Reference

Object Hierarchy and Inheritance in JavaScript - Netscape Online Documentation

ECMA-262 (PDF) - ECMAScript Standard Documentation