Please Note: This document is a little out of date. See my short series of articles at Blogascript - starting with Objects and the Prototype Chain

Again, this article is deprecated. The techniques within are valid, but do not represent best practice.

Object-Oriented Programming in JavaScript

Recently, I have been doing a lot of thinking about the object-oriented side of JavaScript. The depth of the problem isn't something that comes up in many applications, simply because inheritance isn't necessary for a lot of them. The ephemeral nature of JavaScript-based applications means that objects don't necessarily stick around for long, and you may not even need more than one copy of many types of objects.

I've avoided using the word *instance* in this case, as I believe that term applies to a class-based (or classical) paradigm of object oriented programming. JavaScript uses a prototype-based (or prototypal) paradigm of object oriented programming. I'd like to avoid comparing the prototypal paradigm with the classical paradigm, as it seems to muddy the waters.

The Classical Problem

JavaScript has already muddied the waters quite sufficiently by including the **new** keyword in the language. **new** is a construct of classical inheritance, used with classes, which serve as a kind of template for creating objects. **new Animal()** creates a new object of type *Animal*, given that there is a class with that name.

At least, that's what happens in the classical paradigm. What happens in JavaScript is different, due to its prototypal nature. Douglas Crockford describes it succinctly on his page about prototypal inheritance:

> *[JavaScript] has a **new** operator, such that **new f()** produces a new object that inherits from **f.prototype***

There is some amount of misdirection (as Crockford says - I think *indirection* is more fitting) involved in the use of **new** in JavaScript. Since JavaScript does not include classes, we must, instead, use functions which then act like classes when used with **new**.

```
function Animal(p_legs, p_word)
{
	this.legs = p_legs;
	this.word = p_word;
	this.walk = function() { /* do stuff */ }
	this.speak = function() { /* do stuff */ }
}
```

Obviously, this is a pretty simple “class”, but let’s take a look at it. Let’s see what happens when we instantiate **Animal**. By Crockford’s logic, we should get a new object which inherits from **Animal.prototype**. But what is **Animal.prototype**? Where does it come from?

It seems that **Animal.prototype** has members assigned to it via **this**, used within the function body of **Animal**. This adds to the indirection, as we’re not setting **this.prototype.legs**, we’re setting **this.legs**. Logically, it would seem that we’re trying to set a member on the function itself, but it turns out that we’re setting a member on the prototype.

Inheritance is further complicated when trying to emulate a classical paradigm in JavaScript. Since there is no explicit way for a function to **extend** (in the classical way of thinking) another function, odd constructs must be used in order to achieve inheritance (and some of these constructs are not actually inheritance). I will leave further exploration of this pseudo-classical style of object oriented JavaScript up to you, as it is beyond my intent to explore the prototypal nature of JavaScript. Suffice to say it takes a simple concept and turns it on its side, introducing unnecessary code as well as unnecessary complexity to the process of writing object oriented JavaScript.

The moral of the story, of course, is to avoid **new**, and take advantage of the flexibility and simplicity of object-oriented JavaScript.

Object Literals

The easiest way to create an object in JavaScript is simply to do it.

Yes. Really. Just create an object, as a one-off collection of names and values, usable immediately without any further complication.

```
var lion = {
      legs: 4,
      word: ‘roar’,
      walk: function() { },
      speak: function() { }
};
```

And that’s all there is to creating a lion. Simple, right? Use the object literal notation **{ }**, with some properties inside it - names and values are separated by colons, and each pair is separated from the next by a comma.

This is great for one-off objects, and it could be all you ever need, if you tend to work on simple applications that require no repetition of functionality. If, however, you are looking at more complex applications, you won’t want to hard-code each of your objects whenever you need one. You’ll want to avoid code repetition, and be able to create various animals whenever you need them. This is where the prototype comes in.

The Prototype and Object.create()

When a property of an object is accessed, there is a look up process that occurs. The object itself is first examined to see if it contains whichever property is being accessed. If the property is found, then that property is used. If it is not, JavaScript turns to the *prototype chain*.

Each object that contains any inherited properties has a property which represents the prototype. This prototype is simply another object, which contains its own properties, including (potentially) another prototype.

When an object does not have the property that is being accessed, JavaScript looks at its prototype. If the prototype doesn’t have that property, then *its* prototype is accessed, and so on and so forth until the property is found, or there are no more prototypes to search.

Given this model, it is easy to see how an object would inherit properties from another object through the prototype chain. It’s really a very simple concept.

However, many JavaScript engines do not allow us to set the prototype of an object, so we have to use a different method of creating objects which inherit from other objects. This method is **Object.create()**, and it creates a new object which has a prototype equal to its first argument. It may be better explained by example.

Let’s say we wanted to create some animals. Not just the lion we previously created, but several. Maybe a house cat, or a dog, or even a human. We’d want some common functionality

amongst all of these, so we'll use object oriented programming to create some animals.

```
var animal = {
      walk: function()
      {
            var s = '', i = 0;
            for (; i < legs; i++)
            {
                  s += 'step ';
            }
            console.log(s);
      },
      speak: function()
      {
            console.log(word);
      }
}
```

And there we have a generic animal. It can become anything it wants to, really. Now, we just need some more specific animals.

```
var lion = Object.create(animal);
lion.legs = 4;
lion.word = 'roar';

var houseCat = Object.create(animal);
houseCat.legs = 4;
houseCat.word = 'meow';

var dog = Object.create(animal);
dog.legs = 4;
dog.word = 'meow';

var human = Object.create(animal);
human.legs = 2;
human.word = 'hello';
```

Look at all those animals. If we examine the objects, we'll see that each has their own properties for **legs** and **word**, as well as a prototype, which contains **walk** and **speak**.
When we try to make one of our animals speak, we simply call **lion.speak()**. JavaScript looks at **lion**, sees it doesn't have a property **speak**, and checks the prototype. The prototype (**animal**) does have **speak**, so it gets called. In the body of that function, **this** refers to **lion**, and not **animal**. Again, JavaScript checks **lion**, and sees the value for **word**, which then gets logged to

the console.

Old Browsers

Before I continue, a bit of a disclaimer. While all modern browsers implement **Object.create()**, some older ones do not. So, Crockford suggests a bit of a shim. Unfortunately, it uses **new**, but it also means that we don't have to:

```
if (typeof Object.create !== 'function')
{
	Object.create = function (o)
	{
		function F() {};
		F.prototype = o;
		return new F();
	};
}
```

Polymorphism

One of the most important concepts in object oriented design is the concept of polymorphism, or the ability to vary the functionality of objects between inheritance levels. If a function needs to do something different for an object further down the prototype chain than it does for an object at a higher level, then it should be able to.

And because of the prototype chain, this becomes very easy. We can give our **human** a different way of speaking than other animals.

```
human.speak = function()
{
	console.log(this.word + ‘, dude’);
}
```

We now have a human surfer. When we call **human.speak()**, JavaScript looks at **human**, and sees that it does, in fact, have a property called **speak**. So, it gets called as a function, and our human adds ‘, dude’ to his favorite word.

A Step Further

We now have all these great animals to play with. We can make them speak, and we can make them walk. But they're kind of just generic animals. What if you want a **dog** named **spot**, or a

houseCat named **mittens**?

Just make more of them.

```
var spot = Object.create(dog);
var mittens = Object.create(houseCat);
```

You can do this as many times as you like, and all of these will be separate objects which share the properties of **dog**, or **cat**, or **lion**, or **human**, all accessed automatically through the prototype chain.

There is something to be careful about, though. If you change **dog**, then **spot** won't be the same anymore. The prototype will have been altered. Similarly, if you change **animal**, then everything that has **animal** in its prototype chain will be affected.

These changes may not be noticeable if you've taken advantage of polymorphism. If you have made the above change to **human** and made a new **human.speak** property, then changing **animal.speak** will not have any effect on any calls to **human.speak**. Calls to **dog.speak** or **lion.speak**, however, will be affected. This applies to *all* properties - not just those properties that happen to be functions. If we set **dog.legs** to **3**, then all of our dogs will only have three legs.

On Design Patterns

In the above code, I have used the most basic of the design patterns associated with object oriented programming in JavaScript. This is what I've been calling *explicit declaration*. This is the simplest, most straightforward, and (I think) purest form of object creation in JavaScript. However, there are two other design patterns that are perfectly valid: *factories*, and *initialization functions*.

A factory is, intuitively, a function that makes objects. Let's see one in action:

```
function makeAnimal(p_legs, p_word)
{
	var obj = Object.create(animal);
	obj.legs = p_legs;
	obj.word = p_word;

	return obj;
}
```

This is a simple enough pattern, based on the concept of code reuse. In previous examples, I was repeating code to set the necessary values on my new objects, and using three lines every time I created a new object. Simply calling **makeAnimal()** removes that excess code in favor of a simple function call. The other benefit of factories, beyond the savings in lines of code, is the fact that the factory function itself creates a closure around the object creation process. This closure can simulate *private* variables, if you choose to add functions (and not just data) to the new object.

```
function makeAnimal(p_legs, p_word)
{
	var x = 17;
	var obj = Object.create(animal);
	obj.legs = p_legs;
	obj.word = p_word;
	obj.getX = function()
	{
		return x;
	}

	return obj;
}
```

The variable **x** is completely hidden in the closure. However, using this technique also reduces the need for the prototype at all, and tends to lean toward the pseudo-classical paradigm. We might as well be creating an object literal in this factory, or using **new** with a constructor-function.

My favorite solution is the use of initialization functions. I prefer them over explicit declaration and factories because they are contained within the object itself (or its prototype chain, depending on inheritance and polymorphism), and can save a few lines of code.

```
var animal = {
	initialize: function(p_legs, p_word)
```

```
        {
              this.legs = p_legs;
              this.word = p_word;
        },
        walk: function()
        {
              var s = ‘’, i = 0;
              for (; i < legs; i++)
              {
                    s += ‘step ‘;
              }
              console.log(s);
        },
        speak: function()
        {
              console.log(word);
        }
}
```

Creating a new animal is now two lines instead of the three for explicit declaration (though factories still have it beat in this area). **initialize**, however, has the advantage of being part of the object itself. It can thus be modified through polymorphism if necessary, and the interface for object creation will not change (where as you need a new factory for each new type of object).

In any case, with some clever conditionals, each of these techniques can be made to set only the values we need to set for a particular object, saving on memory usage.

Depending on the complexity of your objects, you may find that one of these options is better than other. Don’t be surprised if you find yourself using a different design pattern for each problem you come across.

On Terminology

Toward the beginning of this article, I mentioned that *instance* was an inappropriate term for the objects that result from prototypal inheritance. Indeed, the term is a holdover from the classical paradigm, and does not completely describe what these new objects are. **dog** is not an *instance of* **animal**. Neither is **spot** an *instance of* **dog.**

In the classical paradigm, a new object created from a class has (on its own) the properties defined in the class. However, in the prototypal paradigm (as implemented by JavaScript), an object created with **Object.create** is, instead, its own object with a prototype equal to the first argument of the function. It is a blank object which *inherits* its properties.

And that is where the fundamental differences lie. In the classical paradigm, *classes* inherit from one another, and then objects are created based on those classes. In the prototypal paradigm, objects inherit from other objects, *and that's it*. In the purely prototypal (and less confusing) implementation of object oriented programming in JavaScript, there is nothing similar to a class.

The problem then becomes *"What do we call these new objects that inherit from old objects?"*

Well, *child* seems an obvious choice: *"**dog** is a child of **animal**."* This seems descriptive enough. And this term is used throughout articles on wikipedia that refer to prototypal inheritance.

I would like to suggest reversing the phrasing, though, and simply stating the prototype of the current object: *"**animal** is the prototype of **dog**."*

Good Practice

There is one area of this article that may be shot down as "bad practice", and I would like to acknowledge this while I am able to eloquently word my response.

The creation of **spot** with a prototype of **dog** means that the prototype of **spot** contains data that is not specific to **spot**. This affects the execution of its functions, as it may change at any time. I have heard it said that storing this kind of data in the prototype is not the suggested practice.
I, however, would like to suggest that it is essential to one of the main benefits of prototypal inheritance - flexibility.

The Benefits

In the prototypal paradigm, if you create an object, and you need more of them, you simply make them (or, rather, make objects that inherit from that object). This - *instantly, and without any further initialization* - gives you access to all the data and functionality inside the prototype object. There is no need to create class upon class upon subclass, or even prototype object upon prototype object upon prototype object. You can simply create objects, use them, and then create more like them (and continue the process with those, if you please).

The other main benefit, as I see it, is actually the simplicity. Attempting to re-create a classical paradigm in JavaScript has been a long and arduous process for me, and not completely successful. However, having discovered the correct way of taking advantage of the prototypal paradigm, I can say that it was a great relief. Inheritance became simple - a process of creating more objects. It is honestly mind-boggling how simple it can be when **new** is not involved.

Contact:

Questions and comments can be sent to ryan.kinal@gmail.com. For filtering purposes, please include “OOJS” in the subject line.