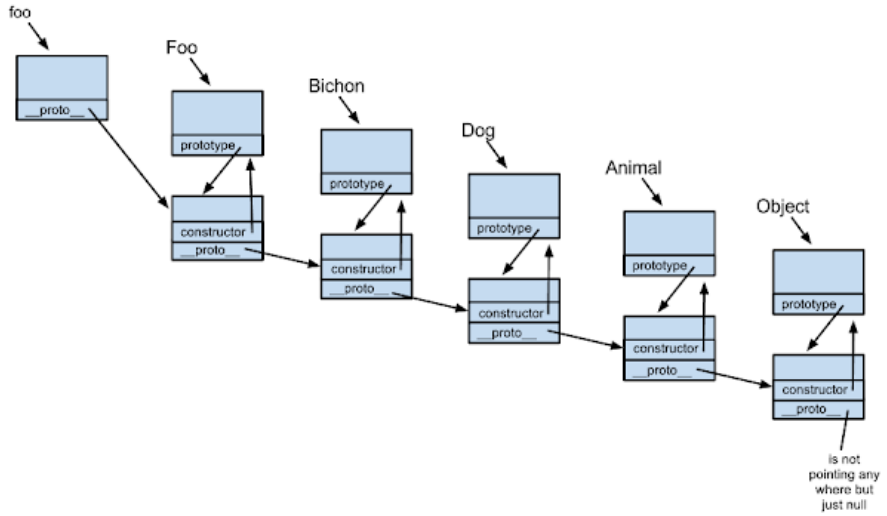
TUESDAY, OCTOBER 2, 2012

JavaScript's Pseudo Classical Inheritance diagram

The following is a chart of JavaScript Pseudo Classical Inheritance. The constructor **Foo** is just a class name for an imaginary class. The **foo** object is an instance of **Foo**.



Note that the **prototype** in **Foo.prototype** is not to form a prototype chain. **Foo.prototype** *points* to some where in a prototype chain, but this **prototype** property of Foo is not to form the prototype chain. What constitute a prototype chain are the **__proto__** pointing up the chain, and the objects pointed to by **__proto__**, such as going from **foo.__proto__**, going up to **foo.__proto__.__proto__**, and so forth, until **null** is reached.

JavaScript's Pseudo Classical Inheritance works like this way: I am a constructor, and I am just a function, and I hold a prototype reference, and whenever **foo = new Foo()** is called, I will let **foo.__proto__** point to my prototype object. So **Foo.prototype** and **obj.__proto__** are two different concepts. **Foo.prototype** indicates that, when an object of **Foo** is created, this is the point where the prototype chain of the new object should point to -- that is, **foo.__proto__** should point to where **Foo.prototype** is pointing at.

In the ECMA-262 Edition 5.1 spec, the term **[[Prototype]]** is used. And that's the same as **__proto__**. It is often mentioned as the "**[[Prototype]]** internal property". And don't confuse this with a function's **prototype** property. One of the key points regarding **[[Prototype]]** is: "All objects have an internal property called **[[Prototype]]**. The value of this property is either **null** or a reference to an object and is used for implementing inheritance."

And now we can see from the diagram why when we inherit **Dog** from **Animal**, we would do:

```
function Dog() {}    // the usual constructor function
Dog.prototype = new Animal();
Dog.prototype.constructor = Dog;
```

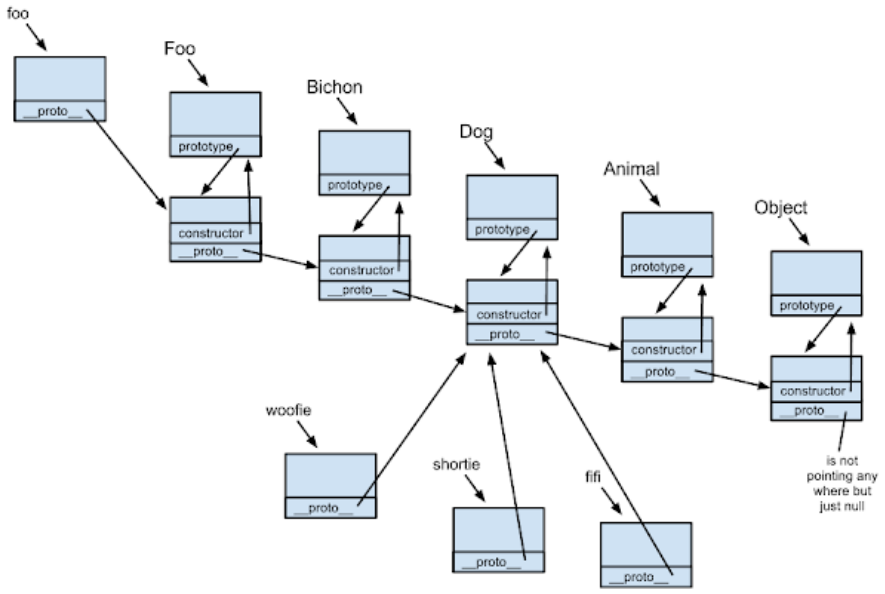
Next, since that new object was created by the **`Animal`** constructor, the new object's **`constructor`** property will point to **`Animal`** (an object created by constructor **`Foo`** will have a **`constructor`** property pointing to **`Foo`**). This constructor property is not **`Dog.prototype`**'s own property -- it is the own property of **`Dog.prototype.__proto__`**, so it is an inherited property. Note that in this case, we actually want the **`Dog.prototype`** object's **`constructor`** property to point to **`Dog`**, and that why we have the second line of code above: **`Dog.prototype.constructor = Dog;`**

Note that we can use an empty function **`F()`** to set up the above relationship as well:

```
function Dog() {}    // the usual constructor function
function F() {}
F.prototype = Animal.prototype;
Dog.prototype = new F();
Dog.prototype.constructor = Dog;
```

Because **`Animal()`** may take a longer time or more complex logic to run, and it can also create properties in that new object that we don't need. A way to set up the **`Dog.__proto__`** to point at **`Animal.prototype`** is what we need, and **`F()`** can already accomplish that.

If there are 3 **`Dog`** instances, they would point to the middle of that long prototype chain. It is still a complete prototype chain, but a shorter one:



Now we can understand why when we add a method to the **`Animal`** class, we would use

```
Animal.prototype.move = function() { ... };
```

That's because when we say

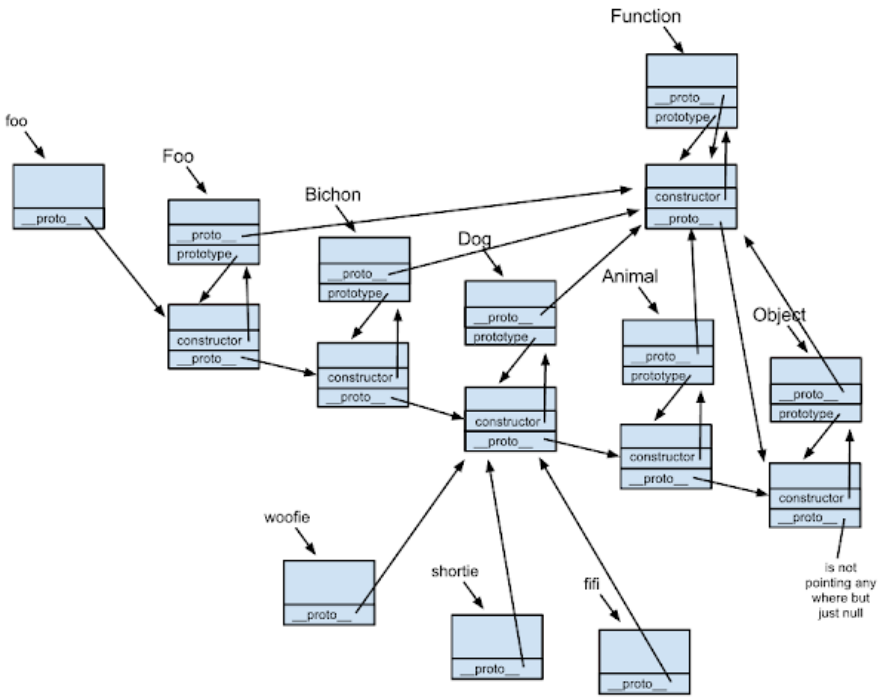
```
woofie.move();
```

If **`woofie`** the object doesn't have the **`move`** method, it will go up the prototype chain, just like any prototypal inheritance scenario, first to the object pointed to by **`woofie.__proto__`**, which is the same as the object that **`Dog.prototype`** refers to. If the method **`move`** is not a property of that object (meaning that the **`Dog`** class doesn't have a method **`move`**), go up one level in the prototype chain, which is **`woofie.__proto__.__proto__`**, or the same as **`Animal.prototype`**. Remember

Note again that this is how prototypal inheritance works, and see how "classical inheritance" is simulated: by the help of prototypal inheritance.

Using the diagram, we can also see the working of **`instanceof`**. For **`foo instanceof Animal`**, it is true, because we take **`foo`** and look at the whole prototype chain, and the **`Animal.prototype`** object is part of that chain. Therefore, it returns true. **`woofie instanceof Animal`** is true for the similar reason: take **`woofie`**'s whole prototype chain, and the **`Animal.prototype`** object is part of that chain. **`woofie instanceof Bichon`** is false because the **`Bichon.prototype`** is not part of that chain. Note that **`woofie.__proto__ instanceof Animal`** is true, the same as **`Dog.prototype instanceof Animal`**, because **`instanceof`** checks for whether the right operand's prototype object is part of the left operand's prototype chain. (Note that **`Dog.prototype instanceof Dog`** used to be true, but it has changed in later implementation of JavaScript: so it will go up to see if **`Dog.prototype`** is part of the chain, but it won't include the object itself (the left operand) to check against **`Dog.prototype`**).

Note that in reality, each constructor function has a **`__proto__`** property as well, and if the **`Function`** constructor is also shown here, a more complete picture is:



Even though **`foo.constructor === Foo`**, the **`constructor`** property is not **`foo`**'s own property. It is actually obtained by going up the prototype chain, to where **`foo.__proto__`** is pointing at. The same is for **`Function.constructor`**. The diagram can be complicated, and sometimes confusing when we see **`Constructor.prototype`**, **`foo.__proto__`**, **`Foo.prototype.constructor`**. Note that Firefox, Chrome, Safari, and node.js support **`__proto__`**, but IE doesn't support it, and it can be obtained by **`Object.getPrototypeOf(foo)`**. (IE 9 or above is needed. Before IE 9, it can be defined as in John Resig's post, and it requires that the **`constructor`** property is set properly.) To verify the diagram, note that even though **`foo.constructor`** will show a value, the property **`constructor`** is not **`foo`**'s own property, but is obtained by following up the prototype chain, as **`foo.hasOwnProperty("constructor")`** can tell.

Posted by Kenneth Kin Lum at 12:34 PM
Labels: javascript, javascript inheritance, prototypal inheritance, prototype, pseudo-classical inheritance

8 comments:

Anonymous June 10, 2013 at 12:23 AM

Amazing. We need more articles like this!

Reply

Anonymous July 31, 2016 at 11:19 AM

Very good. Now I can undestand better.

Reply

Peter Chang January 12, 2017 at 10:17 PM

It is a great great great article, I am struggling in Javascript inheritance, and this article helps me so much to understand it.
thx for writing this

Reply

Peter Chang January 12, 2017 at 10:18 PM

also would u consider to create a git (.js) for this example ?

Reply

Anonymous April 5, 2017 at 1:07 PM

The best explanation in the whole wide web

Reply

jahangir khan April 13, 2017 at 6:03 AM

Snapchat Account

Reply

Anonymous October 20, 2017 at 2:55 AM

I find it hard to follow. But I can see that you were explaining quite good. I would have preferred if you had demonstrated each step with a diagram. It would have made you explanation the best. I hope you would consider it next time.

But now i have to read it like 20 times, to even understand what is going on.

Remember 'A picture is worth a thousand words'.

Reply

FOLLOWERS

Followers (1)

Follow

ABOUT ME

Kenneth Kin Lum

View my complete profile