Internals of JavaScript and inheritance models

Sivaprasad Padisetty (October 2013)

*I broadly classify learning into two levels. Level one is, learn the functionality, so that it can be applied to solve a problem. (e.g.) learn JavaScript or SQL. Level two is to understand the inner functioning of the system. (e.g.) How transactions are implemented by the SQL engine. Very few JavaScript programmers that I spoke to, seem to understand the nuances of JavaScript prototype. I did various experiments to unravel this mystery. This blog is the outcome of that.*

*Nice thing about JavaScript is, it exposes the internals of the objects. This means you can design your own inheritance behavior, even change things dynamically!*

*The intent of this blog is not to give a complete inheritance solution, but to bring out the JavaScript architecture, so that appropriate inheritance can be designed. Warning: \_\_proto\_\_ is deprecated and might not work in all JavaScript engines, however this beautifully illustrates the concepts.*

# \_\_proto\_\_ chain

* The concept of an object in JavaScript is pretty simple, just key value pairs (hashtable). The values can be a property or a function pointer.
* New properties can be added to any object. This is done by “obj.prop1 = …”
* Every object in JavaScript has a special property “\_\_proto\_\_”.
* When a property is accessed and is not found in the current object, it searches in the object pointed by the \_\_proto\_\_ property of the current object. This process continues until it finds the property or \_\_proto\_\_ is null.
* Ideally, there is single root null value. Navigating the \_\_proto\_\_ chain for any object leads here. As I said before, JavaScript provides incredible power, this means you can short circuit and make null for any object’s \_\_proto\_\_.
* When a new property is added to the object, it is always added to the current object. It will not follow the \_\_proto\_\_ chain.

The following lines demonstrate this concept. Try them at [http://repl.it](http://repl.it/). Two objects “a” and “b” are created. “prop1” is added to “a”, it is not available to “b” (it is shown as undefined below). Then prop2 is added to “a.\_\_proto\_\_”, to demonstrate the \_\_proto\_\_ chain. When “a.prop2” is updated, it is added to the “a” and does not travel up the chain. (i.e.) altering “a.prop2” does not change “a.\_\_proto\_\_.prop2”.

|  |
| --- |
| a = {} |
| => {} |
| b = {} |
| => {} |
| a.prop1 = 'value1' |
| => 'value1' |
| a.prop1 |
| => 'value1' |
| b.prop2 |
| => undefined |
| a.\_\_proto\_\_.prop2 = 'value2' |
| => 'value2' |
| a.prop2 |
| => 'value2' |
| a.prop2 = 'new value' |
| => 'new value' |
| a.\_\_proto\_\_.prop2 |
| => 'value2' |
| a.prop2 |
| => 'new value' |

# prototype

In JavaScript, functions are first class objects. This means the function object also has \_\_proto\_\_ property, just like any other object. In addition, the function object in JavaScript has a special property called “prototype”. This enables the function object to behave like a Class. This is very similar to that of a Class object in Ruby.

When a function object is used like a Class, the function plays the role of the constructor. The \_\_proto\_\_ of the newly created object (instance) will point to the prototype property of the function object. This means all the properties that are part of the prototype object are available to the newly created object (because of the \_\_proto\_\_ chain). Here is how inheritance works in JavaScript.

|  |
| --- |
| function Point(x, y) { this.x = x; this.y = y; } |
| Point.prototype.translate = function (x, y) { this.x += x; this.y += y; } |
| var a = new Point(10, 10); |
| a.translate(100, 100); |
| console.log ('After translate x = ' + a.x + ', y = ' + a.y); |
| function Point(x, y) { this.x = x; this.y = y; } |

Properties can be added to the prototype any time. If a new property is added, it is visible to existing objects as well.

|  |
| --- |
| Point.prototype.prop1 = 'value1' |
| => 'value1' |
| a.prop1 |
| => 'value1' |

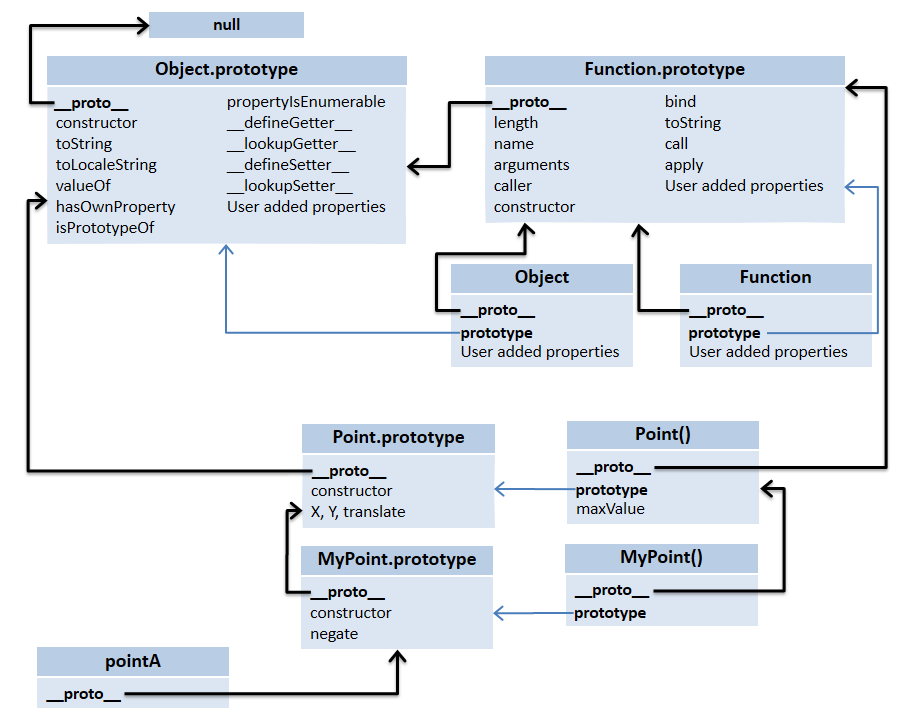
# JavaScript object mystery

The following picture shows how various objects are connected. I did lot of searching around, but could not find a nice comprehensive picture, so I made my own. Hopefully, by end of this blog, the mystery is uncovered, others can benefit from this as well. Each block in the picture is a JavaScript object. Key points:

* Dark black pointer shows the \_\_proto\_\_ chain. Blue arrow is for the prototype pointer
* It is not shown in the picture, the constructor property points to the respective Function object. (e.g.) Function.prototype.constructor === Function
* Object.prototype.\_\_proto\_\_ is a null value, so the \_\_proto\_\_ chain look up stops here. Every \_\_proto\_\_ chain will finally end up here. (unless a null \_\_proto\_\_ is intentionally created)
* The objects “Object”, “Function”, “Point”, “MyPoint” can be logically considered like objects representing a class object.
* Each of these object “Object”, “Function” etc. points to its respective prototype object.
* The properties defined on the class objects (function objects), are like static functions in Ruby.
* “Point” class has a static property of “maxValue”. This can be invoked by Point.maxValue.
* The function object has the prototype property, which points to a different object. This is what is inherited by the class instances (instances properties). These are analogous to Ruby’s instance properties.
* Function is a special object, both \_\_proto\_\_ and prototype point to Function.prototype object. Instances of Function are like classes, they inherit the functionality from Function.prototype, while Function is a function itself! This can be considered as a metaclass, the instances of this is a class.

The following REPL validates the connections between Object/Function.

|  |
| --- |
| Object.prototype.\_\_proto\_\_ |
| => null |
| Function.prototype.\_\_proto\_\_ === Object.prototype |
| => true |
| Object.\_\_proto\_\_ === Function.\_\_proto\_\_ |
| => true |
| Function.\_\_proto\_\_ === Function.prototype |
| => true |
| Function.\_\_proto\_\_ === Function.prototype |
| => true |
| Object.prototype === Object.prototype |
| => true |
| Object.getOwnPropertyNames(Object.prototype) |
| => [ 'constructor', |
| 'toString', |
| 'toLocaleString', |
| 'valueOf', |
| 'hasOwnProperty', |
| 'isPrototypeOf', |
| 'propertyIsEnumerable', |
| '\_\_defineGetter\_\_', |
| '\_\_lookupGetter\_\_', |
| '\_\_defineSetter\_\_', |
| '\_\_lookupSetter\_\_', |
| '\_\_proto\_\_' ] |
| Object.getOwnPropertyNames(Function.prototype) |
| => [ 'length', |
| 'name', |
| 'arguments', |
| 'caller', |
| 'constructor', |
| 'bind', |
| 'toString', |
| 'call', |
| 'apply' ] |



# Inheritance model 1

This follows the model described in <http://phrogz.net/js/classes/OOPinJS2.html>. Once you understand the above layout, \_\_proto\_\_, prototype, it is trivial. Create an instance of the parent class. This object has all the properties of the parent class (because it is an instance of the parent class). Point the prototype property from the child class to this object. Note: prototype can point to any object. This means the instances of the child class, will have all the properties of the parent class, thus achieving inheritance.

Point class has a property called “maxValue”. This is like a static class property in other languages. (i.e.) This can be accessed via Class object, “Point.maxValue” and not via the instances of the class. Point also has the three instance properties X, Y, translate. The function translate will shift the point, by the specified x, y value.

MyPoint is derived from Point. This adds one more instance property called negate. This function negates X and Y values.

Essentially do the following

* Property inheritance is achieved by, ChildClass.prototype = new ParentClass();
* Reset the constructor property for the class using ChildClass.prototype.constructor=ChildClass. Without this, the ChildClass constructor will point to ParentClass constructor.

|  |
| --- |
| function Point(x, y) { this.x = x; this.y = y; } |
| Point.maxValue = 1000; |
| Point.prototype.translate = function (x, y) { this.x += x; this.y += y; } |
|  |
| var a = new Point(10, 10); |
| a.translate(100, 100); |
| console.log ('After translate x = ' + a.x + ', y = ' + a.y); |
| console.log ('maxValue = ' + Point.maxValue); |
|  |
| //Inherit MyPoint from Point |
| function MyPoint(x, y) { Point.call (this, x, y) } |
| MyPoint.prototype = new Point(); |
| MyPoint.prototype.constructor = MyPoint |
|  |
| var b = new MyPoint(20, 20); |
| b.translate(100, 100); |
| console.log ('MyPoint x = ' + b.x + ', y = ' + b.y); |
|  |
| // NOTE: properties defined in the Point is not inheritted |
| console.log ('MyPoint.maxValue = ' + MyPoint.maxValue); |

The output is:

After translate x = 110, y = 110

maxValue = 1000

MyPoint x = 120, y = 120

MyPoint.maxValue = undefined

The limitation with this model is, the properties that are added to the parent class directly (not to the prototype object) are not inherited. In other words, the static properties are not inherited. In the above example, MyPoint.maxValue is undefined.

# Inheritance model 2, Ruby style

An alternate proposal is described below, to fix the above limitation. In fact, the above picture’s connections are based on this model. Following are the steps involved to achieve this:

* ChildClass.prototype.\_\_proto\_\_ = ParentClass.prototype
* ChildClass.\_\_proto\_\_ = ParentClass

Sample script to illustrate this:

|  |
| --- |
| //Define Point |
| function Point(x, y) { this.x = x; this.y = y; } |
| Point.maxValue = 1000; |
| Point.prototype.translate = function (x, y) { this.x += x; this.y += y; } |
|  |
| var a = new Point(10, 10); |
| a.translate(100, 100); |
| console.log ('After translate x = ' + a.x + ', y = ' + a.y); |
| console.log ('maxValue = ' + Point.maxValue); |
|  |
| //Inherit MyPoint from Point |
| function MyPoint(x, y) { Point.call (this, x, y) } |
| MyPoint.prototype.\_\_proto\_\_ = Point.prototype; |
| MyPoint.prototype.negate = function () { this.x = -this.x; this.y = -this.y; } |
| MyPoint.\_\_proto\_\_ = Point |
|  |
| var b = new MyPoint(20, 20); |
| b.negate(); |
| console.log ('MyPoint after negating x = ' + b.x + ', y = ' + b.y); |
| console.log ('MyPoint.maxValue = ' + MyPoint.maxValue); |

The output is:

After translate x = 110, y = 110

maxValue = 1000

MyPoint after negating x = -20, y = -20

MyPoint.maxValue = 1000

# Conclusion

I walked you through how the object system in JavaScript works and then explained two inheritance models. By no means these models are complete, but illustrate the model. Hopefully, this gave you some insights to build your own model.

# References

1. <http://yehudakatz.com/2011/08/12/understanding-prototypes-in-javascript/>
2. <http://phrogz.net/js/classes/OOPinJS2.html>

Explore & Enjoy!

/Siva