by Angus Croll

JavaScript, JavaScript…

Understanding JavaScript Prototypes.

(en Español (http://www.programania.net/diseno-de-software/entendiendo-los-prototipos-en-javascript/), русском (http://interosite.ru/articles/chto-takoe-prototipnoe-nasledovanie-prototipy-obektov-v-javascript), 中文 (http://www.oschina.net/translate/understanding-javascript-prototypes))

JavaScript's prototype object generates confusion wherever it goes. Seasoned JavaScript professionals, even authors frequently exhibit a limited understanding of the concept. I believe a lot of the trouble stems from our earliest encounters with prototypes, which almost always relate to *new*, *constructor* and the very misleading *prototype* property attached to functions. In fact prototype is a remarkably simple concept. To understand it better, we just need to forget what we 'learned' about constructor prototypes and start again from first principles.

What is a prototype?

A prototype is an object from which other objects inherit properties

Can any object be a prototype?

Yes.

Which objects have prototypes?

Every object has a prototype by default. Since prototypes are themselves objects, every prototype has a prototype too. (There is only one exception, the default object prototype at the top of every prototype chain. More on prototype chains later)

OK back up, what is an object again?

An object in JavaScript is any unordered collection of key-value pairs. If it's not a primitive (undefined, null, boolean, number or string) it's an object.

You said every object has a prototype. But when I write *({}).prototype* **I get** *undefined***. Are you crazy?**

Forget everything you learned about the prototype property – it's probably the biggest source of confusion about prototypes. The true prototype of an object is held by the internal *[[Prototype]]* property. ECMA 5 introduces the standard accessor *Object.getPrototypeOf(object)* which to-date is implemented in Firefox, Safari, Chrome and IE9. In addition all browsers except IE support the non-standard accessor *__proto__*. Failing that we can ask the object's constructor for its prototype property.

```
1  var a = {};
2
3  Object.getPrototypeOf(a); //[object Object]
4
5  a.__proto__; //[object Object]
6
7  //all browsers
8  //(but only if constructor.prototype has not been replaced and fails with Object.create)
9  a.constructor.prototype; //[object Object]
```

Ok fine, but *false* **is a primitive, so why does** false.__proto__ **return a value?**

When a primitive is asked for it's prototype it will be coerced to an object.

```
1  //
2  false.__proto__ === Boolean(false).__proto__; //true
```

I want to use prototypes for inheritance. What do I do now?

It rarely makes sense to set a prototype for one instance and only one instance, since it would be equally efficient just to add properties directly to the instance itself. I suppose if we have created a one off object which we would like to share the functionality of an established object, such as Array, we might do something like this (in __proto__ supporting browsers).

```
1  var a = {};
2  a.__proto__ = Array.prototype;
3  a.length; //0
```

But the real power of prototype is seen when multiple instances share a common prototype. Properties of the prototype object are defined once but inherited by all instances which reference it. The implications for performance and maintenance are obvious and significant.

So is this where constructors come in?

Yes. Constructors provide a convenient cross-browser mechanism for assigning a common prototype on instance creation.

Just before you give an example I need to know what this *constructor.prototype* property is all about?

OK. Firstly JavaScript makes no distinction between constructors and other functions, so every function gets a prototype property (built-in function excepted). Conversely, anything that is not a function does not have such a property.

```
1  //function will never be a constructor but it has a prototype property anyway
2  (new Function()).prototype; //[object Object]
3
4  //function intended to be a constructor has a prototype too
5  var A = function(name) {
6  this.name = name;
7  }
8  A.prototype; //[object Object]
9
10 //Math is not a function so no prototype property
11 Math.prototype; //null
```

So now the definition: A function's *prototype* property is the object that will be assigned as the prototype to all instances created when this function is used as a constructor.

It's important to understand that a function's prototype property has nothing to do with it's actual prototype.

```
1  //(example fails in IE)
2  var A = function(name) {
3  this.name = name;
4  }
5
6  A.prototype == A.__proto__; //false
7  A.__proto__ == Function.prototype; //true - A's prototype is set to its constructor's prototype pro
```

Example please?

You've probably seen and used this a hundred times but here it is once again, maybe now with added perspective.

```
1  //Constructor. &lt;em&gt;this&lt;/em&gt; is returned as new object and its internal [[prototype]]
2  var Circle = function(radius) {
3  this.radius = radius;
4  //next line is implicit, added for illustration only
5  //this.__proto__ = Circle.prototype;
6  }
7
8  //augment Circle's default prototype property thereby augmenting the prototype of each generated i
9  Circle.prototype.area = function() {
10 return Math.PI*this.radius*this.radius;
11 }
12
13 //create two instances of a circle and make each leverage the common prototype
14 var a = new Circle(3), b = new Circle(4);
15 a.area().toFixed(2); //28.27
16 b.area().toFixed(2); //50.27
```

That's great. And if I change the constructor's prototype, even existing instances will have access to the latest version right?

Well….not exactly. If I modify the existing prototype's property then this is true, because *a.__proto__* is a reference to the object defined by A.prototype at the time it was created.

```
1  var A = function(name) {
2  this.name = name;
3  }
4
5  var a = new A('alpha');
6  a.name; //'alpha'
7
8  A.prototype.x = 23;
9
10 a.x; //23
```

But if I replace the prototype property with a new object, *a.__proto__* still references the original object.

```
1  var A = function(name) {
2  this.name = name;
3  }
4
5  var a = new A('alpha');
6  a.name; //'alpha'
7
8  A.prototype = {x:23};
9
```

```
a.x; //null
```

What does a default prototype look like?

An object with one property, the constructor.

```
var A = function() {};
A.prototype.constructor == A; //true

var a = new A();
a.constructor == A; //true (a's constructor property inherited from it's prototype)
```

What does instanceof have to do with prototype?

The expression *a instanceof A* will answer true if A's prototype property occurs in a's prototype chain. This means we can trick *instanceof* into failing

```
var A = function() {}

var a = new A();
a.__proto__ == A.prototype; //true - so instanceof A will return true
a instanceof A; //true;

//mess around with a's prototype
a.__proto__ = Function.prototype;

//a's prototype no longer in same prototype chain as A's prototype property
a instanceof A; //false
```

So what else can I do with prototypes?

Remember I said that every constructor has a *prototype* property which it uses to assign prototypes to all instances it generates? Well that applies to native constructors too such as Function and String. By extending (not replacing!) this property we get to update the prototype of every instance of the given type.

I've used this technique in numerous previous posts to demonstrate function augmentation. For example the tracer utility (https://javascriptweblog.wordpress.com/2010/06/01/a-tracer-utility-in-2kb/) I introduced in my last post needed all string instances to implement *times*, which returns a given string duplicated a specified number of times

```
String.prototype.times = function(count) {
return count &amp;lt; 1 ? '' : new Array(count + 1).join(this);
}

"hello!".times(3); //"hello!hello!hello!";
"please...".times(6); //"please...please...please...please...please...please..."
```

Tell me more about how inheritance works with prototypes. What's a prototype chain?

Since every object and every prototype (bar one) has a prototype, we can think of a succession of objects linked together to form a prototype chain. The end of the chain is always the default object's prototype.

```
a.__proto__ = b;
b.__proto__ = c;
c.__proto__ = {}; //default object
{}.__proto__.__proto__; //null
```

The prototypical inheritance mechanism is internal and non-explicit. When object *a* is asked to evaluate property *foo*, JavaScript walks the prototype chain (starting with object *a* itself), checking each link in the chain for the presence of property *foo*. If and when *foo* is found it is returned, otherwise undefined is returned.

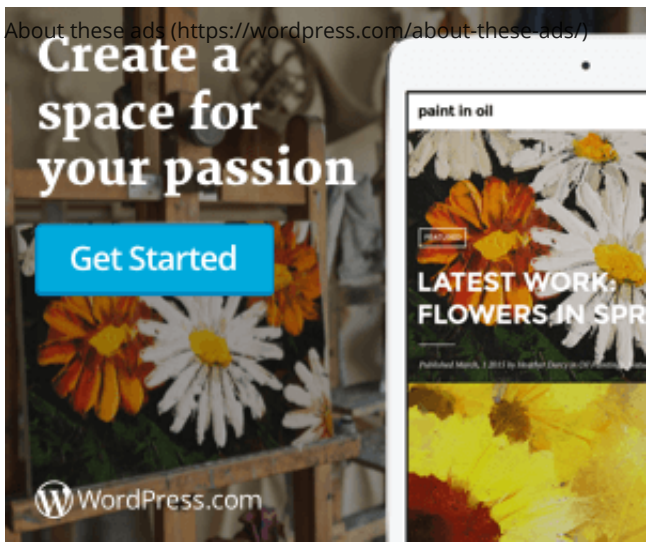
What about assigning values?

Prototypical inheritance is not a player when property values are set. *a.foo = 'bar'* will always be assigned directly to the foo property of a. To assign a property to a prototype you need to address the prototype directly.

And that about covers it. I feel I have the upper hand on the prototype concept but my opinion is by no means the final word. Please feel free to tell me about errors or disagreements.

Where can I get more information on protoypes?

I recommend this (http://dmitrysoshnikov.com/ecmascript/chapter-7-2-oop-ecmascript-implementation/) excellent article by Dmitry A. Soshnikov



JavaScript (https://javascriptweblog.wordpress.com/category/javascript/)

Angus Croll
June 7, 2010December 3, 2015

constructor /
Inheritance /
prototype

110 thoughts on “Understanding JavaScript Prototypes.”

Nick Fitzgerald says:
June 7, 2010 at 10:24
Another well researched post, Angus! One nitpick, you say:

“Prototypical inheritance is not a player when property values are set. a.foo = ‘bar’ will always be assigned directly to the foo property of a. To assign a property to a prototype you need to address the prototype directly.”

It is possible (though, not cross-browser, I have only tested with FF and Chrome, IE definitely won’t) to define setters and getters for properties via

```
1 Object.prototype.__defineGetter__
2 Object.prototype.__defineSetter__
```

As always, MDC has a great article:
https://developer.mozilla.org/en/Core_JavaScript_1.5_Guide/Working_with_Objects#Defining_Getters_and_Setters

Reply

Angus Croll says:
June 7, 2010 at 11:01
Thanks Nick – Yeah I know, ECMA 5 is standardizing get and set properties too. But I wanted to keep it simple and also caution against having one instance update the prototype. Nice point though

Reply

Vyacheslav Egorov (@mraleph) says:
June 30, 2012 at 13:20
and you don’t even need a setter. e.g. if you add a non-writable property foo to the prototype of obj that will prevent obj.foo = 10 from succeeding unless obj already has foo.

Dmitry A. Soshnikov says:
June 8, 2010 at 03:10
This is a good overview.

If and when foo is found it is returned, otherwise null is returned

undefined is returned, but not *null* — http://dmitrysoshnikov.com/ecmascript/chapter-7-2-oop-ecmascript-implementation/#get-method

P.S.: It is good nevertheless to mention a list of used/additional literature on which your article is based

For details: http://dmitrysoshnikov.com/ecmascript/chapter-7-2-oop-ecmascript-implementation/

Dmitry.

Reply

Angus Croll says:
June 8, 2010 at 08:49
Dmitry, Yes good correction. Also thanks for including you article references

Apologies for not mentioning your article You'll notice from my other posts I am always happy to credit sources . I do not think my article is based on your article but I did make use use of your perfect definition of object and I gained inspiration from re-reading your article while writing mine, and it reminded me to cover some points. Thank you.

BTW I would encourage all readers to check out Dmitry's entire ECMA series. It's fantastic.
http://dmitrysoshnikov.com/2010/04/

Reply
Oral Dalay (@perfectionkills) says:
November 4, 2012 at 15:15
here is the updated url for Dimitry's article
undefined is returned, but not null — http://dmitrysoshnikov.com/ecmascript/chapter-7-2-oop-ecmascript-implementation/#codegetcode-method

Reply
Pingback: links for 2010-06-27 « Treat with Jermolene
Pingback: Understanding JavaScript Prototypes. « JavaScript, JavaScript « Guide Weblog
Pingback: Understanding JavaScript Prototypes. « JavaScript, JavaScript « Information Site Weblog
Witek Baryluk says:
June 30, 2010 at 15:04
Hey. Thanks for this great article. I have been using JS many years as small scripting language, but now I'm creating really big project fully in JS, and wanted to know what is really going in all details in it. Could not find anything useful, wanted to start reading Mozilla Documentation, and noticed your article on reddit. Nice. Saved me some time.

How about setters and getters? How they cover assigning properties?

Reply
Angus Croll says:
June 30, 2010 at 15:19
Hi Witek, glad this was helpful

>>How about setters and getters? How they cover assigning properties?

Do you mean in respect to prototypes, or is this a suggestion for another blog topic?

Reply
Witek Baryluk says:
June 30, 2010 at 18:55
Yes what is their connection (or how they could be connecte) with prototype chains?

Citing: "Prototypical inheritance is not a player when property values are set. a.foo = 'bar' will always be assigned directly to the foo property of a. To assign a property to a prototype you need to address the prototype directly."

But what if foo is a setter in a prototype? Well do not know how to define it, but could be useful.

Angus Croll says:
July 1, 2010 at 08:59
Nick Fitzgerald mentioned this in an earlier comment

In some browsers you can do this:
A.prototype.__defineSetter__("b", fn);

(https://developer.mozilla.org/en/Core_JavaScript_1.5_Guide/Working_with_Objects#Defining_Getters_and_Setters)

By the way, accessors have been standardized in ECMA5 (see http://dmitrysoshnikov.com/ecmascript/es5-chapter-1-properties-and-property-descriptors/#named-accessor-property)

I'm not sure I recommend this approach for prototypes though

Pingback: Understanding JavaScript Prototypes. « JavaScript, JavaScript « Blog Weblog
Pingback: Understanding JavaScript Prototypes. « JavaScript, JavaScript « Guide Weblog
Pingback: Understanding JavaScript Prototypes. « JavaScript, JavaScript « Info Weblog
Pingback: Understanding JavaScript Prototypes. « JavaScript, JavaScript | Uber Daily
Timothy Fortune says:
September 9, 2010 at 08:50
Great post! I've used this numerous times just trying to figure all this out.

I do have one thing that is confusing me a little, though. Your description of how instanceOf is defined is confusing to me a little. Wouldn't it be better to define

a instanceOf A

as being true whenever A.prototype falls in a's prototype chain.

Consider the following scenario. Assume you have created a class called USADate which derives from Date, and further assume we have created instances of each called respectively aUSADate and aDate. According to your definition, wouldn't the following be true, even though it really isn't?

aUSADate instanceOf Date,

I may just not be understanding correctly, since I am new to this. Any explanation would be appreciated, and again, thanks for the great blog post.

Reply

Angus Croll says:
September 9, 2010 at 09:37
Hi Timothy – thanks for the nice comments

I think your definition "whenever A.prototype falls in a's prototype chain" is pretty much word for word the same as mine.

But in your example:

```
1 | aUSADate instanceOf Date
```

is true, assuming that by "derives from" you mean USADate inherits from Date's prototype

A simpler example of the same thing (remember Object's prototype is at the top of all prototype chains):

```
1 | function A() {};
2 | var a = new A();
3 | a instanceof Object; //true
```

Reply

Waylon Flinn (@waylonflinn) says:
December 13, 2012 at 07:38
Great article. I really brings lots of information about prototypes together. I think I agree with Timothy on this point, though.

You seem to be suggesting that the operation for determining when instanceOf is true is that you take the prototype for 'a' and look for it in the prototype chain of 'A'. The converse of this operation is what's actually true. You look for the prototype of 'A' in the prototype chain of 'a'.

Here's a quote from MDN

The instanceof operator tests whether an object has in its prototype chain the prototype property of a constructor.

The object here is 'a'. The constructor is 'A'. (I'm sure that's obvious to you, but I wanted to include it for other readers)

I'd like to suggest the following edit:

"The expression a instanceof A will answer true if A's prototype property can be found somewhere in a's prototype chain."

Angus Croll says:
December 13, 2012 at 07:49
Hey Waylon – you're right I should clarify that definition – also revisiting my example – it's kind of lame!

Thanks!

Pingback: Prototypal Inheritance in JavaScript « Rupesh kumar Tiwari
Pingback: Cor Cinza » JavaScript for the PHP programmer
AndyN02 says:
October 6, 2010 at 05:53
cool post. there is something about the prototype thing that blows my head off If my class inherits from another class then there is no valid way anymore to add getters and setters (with valid i not mean things like "__defineGetter__" , "__defineSetter__","__proto__"). If im not totaly of the road getters an setters should be defined like this:

```
var car = function(){};
new car();
car.prototype =
{
_a:0,
get a(){return this._a},
set a(x){this._a=x}
};
```

but if i try to inherit cars

```
var audi = function(){};
```

```
new audi();
audi.prototype = new car;
// that clearly couldnt work :(
audi.prototype =
{
_b:10,
get b(){return this._b},
set b(x){this._b=x}
};
```

there are some workarounds:

```
extend(audi,car);
//workaround 1 (Firefox and Crome only)
function extend(child, supertype){
child.prototype.__proto__ = supertype.prototype;
}


 //workaround 2 (all browsers but dirty!)
function extend(child, super){
for (var property in super.prototype) {
if (typeof child.prototype[property] == "undefined")
child.prototype[property] = super.prototype[property];
}
return child;
}
```

is there another way?

Reply

Angus Croll says:
October 8, 2010 at 08:40
Thanks Andy

Check out this post for an example on inheritance using prototypes – but beware JavaScript is not Java (etc.) and classical inheritance is less suited to the language

Reply

Richard Bobinski says:
February 16, 2011 at 09:07
I just started looking at jPaq's Color object, and must say that it is quite interesting what Chris West did. It seems that he has made a way of keeping private variables private. For anybody that is interested, I would definitely take a look at how he implemented this object.

Reply

Pingback: JavaScript is Awesome and Ubiquitous « Cootcraig's Blog

heswell says:
April 14, 2011 at 10:32
IE9 also has Object.getPrototypeOf(object). Last time I checked, IE9 had the most complete implementation of ECMA5 features of any released browser (and IE10 adds strict mode)

Reply

Pingback: Javascript – How Prototypal Inheritance really works | Vjeux

Rahul Mohan says:
June 7, 2011 at 03:34
Excellent article. Thanks a million! I have been reading a lot of stuff to get an understanding of the prototype. This is where my quest ends. If you are interested in the philosophical reasoning behind prototypical object orientation as opposed to the popular classical object orientation then this paper is a must read.

Reply

Angus Croll says:
June 9, 2011 at 14:24
@Rahul glad to help – that's quite a paper! Plato and Aristotle are cited – I look forward to reading it properly

Reply

Brian P says:
June 17, 2011 at 10:38
As a relatively new javascript programmer, I really appreciated this article. Thank you for keeping it so clear and uncluttered with all the exceptions or possibilities that usually come with programming. The question/answer style made the concepts very easy to follow.

Reply

Pingback: Code snippet « monkeylearns
Anonymous says:
June 29, 2011 at 16:24
Thanks for your explanation, this really helps me understand prototype better.
You did understand what confused so many programmers, the prototype property of a function. I read several books on javascript, none of them gets the points on this.

Reply
Pingback: Link-urile săptămânii 27 iunie – 3 iulie | Staicu Ionuţ-Bogdan
ManiKanta G says:
July 24, 2011 at 12:14
Nice & very insightful article about prototypes.

I want to clarify my understandings a little but at the core level.

Citing two statements from above article:

> Every object has a prototype by default

and

> OK. Firstly JavaScript makes no distinction between constructors and other functions, so every function gets a prototype property. Conversely, anything that is not a function does not have such a property.

When you mean, only functions will have prototype property, both Object and Functions (which are actually Objects, of course) will have prototypes, but it is just functions that will have 'prototype' property accessible for explicit use. But objects (var a = {}) will also have implicit, non-accessible prototype object. Is my understanding correct?

And if I m correct and we can't get the prototype of an object (not a function), means we can't make a object inherit from other object. Correct?

Thanks once again
ManiKanta

Reply
Angus Croll says:
July 24, 2011 at 13:00
Hi Mani,

Thanks for the nice words!

To clarify, all objects have a prototype (an internal property, represented in some browsers by the "__proto__" identifier) but only functions have a property named "prototype". The "prototype" property does not represent the prototype of the function, it references the prototype that will be assigned to instances created when that function acts as a constructor.

```
var myConstructor = function() {};
var myInstance = new myConstructor();

//('__proto__' not available in all browsers)
myInstance.__proto__ === myConstructor.prototype; //true
```

Hope this helps!
Angus

Reply
ManiKanta G says:
July 24, 2011 at 13:11
Thanks Argus. Thanks for clarifying again and that means I understood it correct.

And about my second question:

> And if I m correct and we can't get the prototype of an object (not a function), means we can't make a object inherit from other object. Correct?

I came to know the answer from one of your other post: https://javascriptweblog.wordpress.com/2010/03/16/five-ways-to-create-obejcts-part-2-inheritance/ – 3. Constructor using Object Literal way.

Thanks
ManiKanta

Angus Croll says:
July 24, 2011 at 14:07
OK, sorry I misread your original question slightly.

Also note that as of ES5 you can use Object.create to make such inheritance a little easier

```
//won't work in IE<9
var myConstructor1 = function() {};
myConstructor1.prototype.prop1 ='apple';

var myPrototype2 = Object.create(myConstructor1.prototype);
myPrototype2.prop2 = 'pear';

var myConstructor2 = function() {};
myConstructor2.prototype = myPrototype2;

var myInstance2 = new myConstructor2;
myInstance2.prop1; //apple
myInstance2.prop2; //pear
```

Anonymous says:
August 14, 2011 at 07:56
Very interesting….thanks!

Reply
Pingback: #JavaScript wait untill a function/method finished executing « Gert-Jan Jansma
Pascal Klein says:
September 21, 2011 at 16:30
Really good article. Now I think I am finally getting the hang of it. Thanks a lot!

Reply
Pingback: 理解JavaScript原型 - 博客 - 伯乐在线
Pingback: Closures, Javascript, and Objects – Part II | In the trenches
Pingback: 2011 December : What I'm Reading
Anonymous says:
January 17, 2012 at 15:05
Excellent article that clarifies a lot!

Reply
Sarfraz Ahmed says:
February 10, 2012 at 23:32
Good effort but I don't think even this article makes much sense to beginners or even intermediate javascript developers. This is yet another confusing article.

Let's forget about complexities and be as simple as this guy has done:

http://timkadlec.com/2008/01/using-prototypes-in-javascript/

This is all one needs to know without going into any complexities and reading all over again and again or searching elsewhere to dig into the topic further.

Reply
Pingback: Javascript: Using Prototype « andrewBridge
onemhz says:
April 9, 2012 at 09:34
I'm working with a javascript library, and I was getting an error that a method is undefined. It appears that it *should* be defined in the function's .prototype.

eg

```
ClassWrapper = {
var someObject = function (inputs) {
…
}

someObject.prototype = {
someMethod: function someMethod_internal() {
…
}
…
}
};
```

Can you clarify when the .prototype comes into play? It doesn't seem that the prototype is being recognized, because all of the methods it defines are undefined.

Reply
Jose_X says:
April 9, 2012 at 14:20
onemhz, I think your syntax is wrong. I am not an expert, but here is what I see.

[Note, you may want to skip the first two examples and maybe the third as well if all you want is to see overloading in action. Since there is a syntax error and your naming doesn't match OOP usage, I had to make assumptions about what you wanted to do. First assumption is that ClassWrapper is an ordinary object. I ignore someObject because I can't figure out what you want exactly, but I give example of .prototype in action. Second assumption is that ClassObject is a Function object and not a mere Object object. I again ignore someObject. Finally, at the end I show overloading where obj1.method1 is different than obj2.method1 even though both obj1 and obj2 came from class. The trick is to change the .prototype of class before creating each object.]

Assumption 1:

ClassWrapper is defined using literal object notation (so it is an object), but then you add a "var someObject = [value]" inside rather than "someObject: [value]". This is a syntax error. If you fix that syntax (eg, use the colon and drop the "var"), then someOjbect is no longer a global variable but an object variable of the object ClassWrapper so you must address its prototype as ClassWrapper.someObject.prototype.

Here is an example showing prototype working properly. You can copy/paste into your browser location bar to run it (I use and tested it on firefox only):

javascript:xx={x:function(a){alert(a);}}; xx.x.prototype={t:7}; var y=new xx.x(8); alert(y.t);

What we see is that xx is an object (like your ClassWrapper) with a property named x. x holds as its value a function that displays the variable you pass into it. Since xx.x is a function, we can use "new" on it, and we can also give it a prototype object. The above line does the latter. It sets the "prototype" property to an object that has one property called "t" assigned the value "7". Now, instead of using new on xx (which I can't do since it is an ordinary object and not a function object), I use it on xx.x, providing the expected parameter that will be displayed. This is why the first alert box shows "8". Finally, we verified that the prototype of the xx.x function does work as expected by displaying the value of y.t (a "7" should have been displayed in the second alert).

Assumption 2:

Now, let's reset and assume that you intended ClassWrapper to be a function instead of an ordinary object (ie, you intend to use it with "new"). We also then use ordinary statements inside the function definitions. Having xx play the role of ClassWrapper and making t a function as well:

javascript:xx=function (b) {this.x=function(a){alert(a);alert(b);}}; xx.prototype={t:function (c) {alert(c);}}; var y=new xx(3); y.x(11); y.t(9); xx.prototype={z:function (d) {alert(d);}}; y.t(22); alert(y.z); w=new xx(44); w.z(33); alert(w.t);

OK, we create xx as a function. When we create y as new xx(3), y is given a property x defined as a function that alerts twice. The first alert is whatever you give to x when you call it, and the second alert is that 3. So we next show that y does have an x property and call it with 11. This means x is called and will alert 11 and then alert 3. Next, we show that since y was created from xx when its prototype included t, y also now can access t. We test this by calling y.t(9) and observe that 9 is displayed. We now change the xx.prototype property. We note that y still can access t and can't access z; however, a new variable w=new xx(44) can access z and can't access t. So the full sequence of alerts is: 11, 3, 9, 22, undefined, 33, undefined.

Overloading Scenario:

Remember that this is not Java or ordinary OOP.

Do you want to give a fresh example of what you want to do with .prototype? Your use of WrapperClass and someObject doesn't really follow how you create Java classes and objects. In fact, you can avoid using .prototype altogether if you don't care about overloading. Just create the methods inside the function definition, for example:

javascript: class= function (initparam) {this.method1=function () {alert("method1 called");}; }; obj=new class("blah"); obj.method1();

If you want to enable overloading for class, then:

javascript: class= function () {}; class.prototype={method1: function () {alert("method1 called at version 1");}}; obj1=new class(); obj1.method1(); class.prototype={method1: function () {alert("method1 called at version 2");}}; obj2=new class(); obj2.method1(); obj1.method1(); obj2.method1();

Here obj1 always prints version 1 while obj2 always prints version 2.

Note that I trashed the original .prototype object (to be garbage collected). We could have saved it for use later in creating other objects like obj1. However, if we had simply set class.prototype.method1 for obj2, then we would be replacing the same method1 value that obj1 uses so obj1 would no longer use version1:

javascript: class= function () {}; class.prototype={method1: function () {alert("method1 called at version 1");}}; obj1=new class(); obj1.method1(); class.prototype.method1= function () {alert("method1 called at version 2");}; obj2=new class(); obj2.method1(); obj1.method1(); obj2.method1();

This prints version 1, 2, 2, 2 instead of 1, 2, 1, 2

Inheritance Scenario:

[Jump to here]

And if you want to use subclassing, try this style where a new subclass is created by calling: Inherit(newclass, superclass, publicmethodsinnewclass); and the methods passed in are what you want to view as the official public interface of overloadable methods.

***

```
Inherit = function (newclass, superclass, methods) {
var o={};
for (var m in superclass.prototype) {
o[m]=superclass.prototype[m];
}
for (var m in methods) {
o[m]=methods[m];
}
newclass.prototype=o;
};
inc=0;

S1 = function (initparams) {
//do something with initparams
this._method1 = function () {alert (++inc+" private method1 in S1");};
};
S1.public1 = function () {alert (++inc+" public1 in S1");};
Inherit(S1, Object, {method1: S1.public1});
S2 = function (initparams) {};
S2.public22 = function () {alert(++inc+" public22 in S2");};
S2.public33 = function () {alert(++inc+" public33 in S2");};
Inherit(S2, S1, {method2:S2.public22, method3:S2.public33});
S3 = function (initparams) {
this._method1 = function () {alert (++inc+" private method1 in S3");};
};
S3.public111 = function () {alert(++inc+" public111 in S3");};
S3.public222 = function () {alert(++inc+" public222 in S3");};
S3.public444 = function () {alert(++inc+" public444 in S3");};
Inherit(S3, S2, {method1:S3.public111, method2:S3.public222, method4:S3.public444});

obj1=new S1();
obj2=new S2();
obj3=new S3();
obj1._method1(); //"1: private method1 in S1"
obj1.method1(); //"2: public1 in S1"
//obj1.method2(); //error
//obj1.method3(); //error
//obj1.method4(); //error
//obj2._method1(); //error
obj2.method1(); //"3: public1111 in S1"
obj2.method2(); //"4: public2222 in S2"
obj2.method3(); //"5: public3333 in S2"
//obj2.method4(); //error
obj3._method1(); //"6: private method1 in S3"
obj3.method1(); //"7: public11 in S3"
obj3.method2(); //"8: public22 in S3"
obj3.method3(); //"9: public3333 in S2"
obj3.method4(); //"10: public44 in S3"
***
```

Reply

Jose_X says:
April 10, 2012 at 18:36
onemhz (continuing from prior subthread):

Jose_X>> I decided (in the Inheritance Scenario section) that perhaps you were trying to use .property to simulate public method overloading

Sorry, ".property" should be ".prototype".

Also, I should add that if you copy/paste the code to run it, you may have to change the quotation marks to get the code to work.

FWIW, I embedded the code above in a wrapper html page I had laying around in order to test it. If everything is working, you should see 10 alert boxes pop up as described above.

As a further exercise, we can create a new class S4 by subclassing S2. We can then add new public methods, say, public5555 and public6666 and overload any of the 3 public methods S2 has. So we could overload public22, public33, or public1 (which was inherited from S1). To carry this out, we might call

Inherit(S4, S2, {public5555: … , public6666: …, public22: …});

where we have overloaded public22 and where the "…" represent an inline function or the name of an existing function (such as the ones I created within S1, S2, and S3 in order to keep them out of the way and quasi-hidden).

To be clear, "Inherit" is not a javascript function or a part of any standard. I made this up. [Its implementation is not complex and almost surely not original, probably influenced subconsciously by something I may have read here or elsewhere.]

Adding multiple inheritance may not be that difficult, but I lost interest in going further.

Also, note that while Java enforces hiding and many of the OOP features, here we have to simply agree not to bypass the contract if we want to simulate the OOP. We can always in javascript fairly simply clobber and replace any private of public method. There is no security (not even through obscurity). Inherit and the approach above is useful if you like the OOP style.

Finally, I want to say that I went to this "trouble" as an exercise for myself because I found this article by Angus Croll interesting and something I had not looked at much before. If you want me to re-explain any part of this, ask. If you find a mistake, you can post a fix to help me and other readers.

Reply

Pingback: Just Enough JavaScript – Prototype and Inheritance | Dapeng Li

Phil says:

May 20, 2012 at 13:44

Great post. I feel like you are reading my mind and reacting appropriately as I read the post. Neat!

Reply

Pingback: "prototype" in jQuery fn? + plain javascript code inside a jquery function? | Easy jQuery | Free Popular Tips Tricks Plugins API Javascript and Themes

Tap says:

June 7, 2012 at 07:58

thx Angus

Reply

ggalmazor (@ggalmazor) says:

June 18, 2012 at 05:44

Thanks for putting a link to our spanish translation!

Reply

Pingback: Pregunta: ¿Dónde están las clases en Javascript? (I) « Newbe forever

Pingback: Dónde están las clases en Javascript (II) « Newbe forever

Julia says:

June 30, 2012 at 14:25

Great article – soooo helpful. Thanks so much

Reply

Dave Stibrany says:

June 30, 2012 at 16:53

Great article! One of the best writings I've found on the prototype topic.

In case any one wants to know more, I found Cody Lindley's book 'JavaScript Enlightenment' really good when talking about prototypes.

Reply

Pingback: Understanding JavaScript Prototypes. | Software languages and frameworks | Scoop.it

Quido Hoekman (@qhoekman) says:

July 1, 2012 at 14:00

Thanks, also really helpful!! I was able to make something like this with you article: http://pastebin.com/hr3BQWwv . It's a sort of anonymous module system for Javascript.

Reply

Pingback: Prototypal vs. Functional Inheritance in JavaScript | Art & Logic Software Development Blog

Pingback: Prototypal vs. Functional Inheritance in JavaScript | Art & Logic Software Development Blog

bartek says:

July 15, 2012 at 14:36

great article;)

Reply

WK_of_Angmar says:

July 30, 2012 at 03:01

(a's constructor property inherited from it's prototype) should be:
(a's constructor property inherited from its prototype)

Reply

Pingback: The NetStat Blog » Blog Archive » Understanding JavaScript Prototypes. | JavaScript, JavaScript…

Marek Karwowski (@iEvaluation) says:

August 13, 2012 at 18:08

Another good post! Well done and thanks! Have just read it and decide to update my JavaScript Confusing Bits. Cheers.

Reply

Pingback: Introduction to Object-Oriented Javascript | Gelblog

Pingback: Fluff 2012 August Notes on Advanced Javascript « TechnoBuzz

Adio says:

September 10, 2012 at 05:30

Very good explained …

Reply

Andreas Boehrnsen (@deepflame) says:

September 14, 2012 at 01:36

Thanks a lot. That was very helpful. I liked your approach of explaining the whole thing in a dialogue.

Reply

binarymist says:

September 15, 2012 at 21:03

@Angus.
This line was the key for me:
this.__proto__ = Circle.prototype
This post is not only an excellent tutorial, but also something I'll use as a reference.
terse, concise and well explained.
Keep them coming.

Reply

Richeve Bebedor says:

September 27, 2012 at 18:22

You said:
"An object in JavaScript is any unordered collection of key-value pairs. If it's not a primitive (undefined, null, boolean, number or string) it's an object."

In some javascript interpreters/engines like the Chrome v8, (I don't know for all interpreters/engine)

null is an object

Try it in your chrome console type:

typeof null

And it will return to you

"object"

Thanks!

Reply

Richeve Bebedor says:

September 27, 2012 at 18:43

Erratum again

You said:
"//function will never be a constructor but it has a prototype property anyway
Math.max.prototype; //[object Object]"

This is temporally incorrect. Any function can be a constructor in the near future if they are instantiated and represented as an object using the keyword "new".

Example:

```
var a = function(){ return "Hello world."; };
var b = new a();
typeof b.constructor; // This will return "function"
b.constructor(); // This will return "Hello world"
b.constructor === a // This is true.
```

Function inside variable a is not a constructor. But it becomes a constructor of object b because we instantiated and represented it as an object.

Correct me if I'm wrong

Thanks!

PS: You can also do this:

```
new (function(){})
```

It will create an anonymous object

Reply

Pingback: JavaScript Properties « Binarymist
Pingback: Understanding JavaScript Prototypes. | JavaScript, JavaScript… | codingfan
Pingback: JavaScript Inheritance « Steven Lasch
Pingback: hojasweb » Herencia en JavaScript
Anonymous says:
October 30, 2012 at 06:48
awesome article. helped me a lot to clear my confusion.

Reply
Emelie says:
November 10, 2012 at 01:31
Thank you SO much for this article! You really saved me.

Reply
Pingback: Javascript: Using Prototype | Andrew H Bridge
Evaldas Ilginis says:
December 26, 2012 at 09:50
The most important sentence, that explained everything: ” It’s important to understand that a function’s prototype property has nothing to do with it’s actual prototype.”
Thank you very much.

Reply
10kloc says:
December 30, 2012 at 12:11
Reblogged this on 10K-LOC and commented:
An excellent post explaining JavaScript’s Object creation mechanism via prototype.

Reply
Siraris says:
January 4, 2013 at 20:54
In your example you say that Math.max.prototype will return the prototype object of the max function. When I log Math.max.prototype it says undefined, while Math.max.constructor returns the Function object. Is this a mistake?

Reply
Angus Croll says:
January 4, 2013 at 22:57
Good catch! Built-ins don’t have the internal [[construct]] property so having a prototype property is unnecessary.

I will update the text shortly. Many thanks for pointing out my error!

Reply
Anonymous says:
January 14, 2013 at 04:46
I’ve also got confused by this line. Math.max is not a usual function indeed and it is not a constructor. new Math.max() yields to error ‘Math.max is no a constructor’.

Anonymous says:
January 14, 2013 at 04:48
I’ve also got confused by this line. Internal functions are not usual indeed in that they are not constructors. ‘new Math.max()’ expression yields to ‘Math.max is not a constructor’ error.

Angus Croll says:
January 14, 2013 at 09:21
You’re right – oversight on my part. Most built-in functions don’t have the internal [[construct]] property, so they don’t get a prototype property and they won’t work with `new` . Updated the text.

Eric Turner says:
January 5, 2013 at 17:07
Great article! I’m a very visual person, so I had to draw a picture to make sense of it.
https://www.dropbox.com/s/nh4a4zicnzwa9s6/javascript_prototype_diagram.png

Reply
Anonymous says:
January 14, 2013 at 06:43
Very nice illustration! But the line foo.__proto__ == Function.prototype seems to be wrong to me.

Reply
Pingback: AMD & RequireJS | MakingSensers
Pingback: Wi Fi Horizon, use copoun inside
ateev says:
February 24, 2013 at 00:30
Reblogged this on Ateev and commented:
JavaScript prototypes cant be explained any better

Reply

Pingback: JS Inheritance in 15mins or Less | A Void: The Infinite
Pingback: JavaScript: Required Reading | James on JavaScript
Pingback: JavaScript Interview Questions: Arrays | Kevin Chisholm - Blog
Pingback: JavaScript Interview Questions: Object-Oriented JavaScript | Kevin Chisholm - Blog
Pingback: Understanding Javascript Prototypes | the pluralsight blog
Sukhvir says:
March 20, 2013 at 05:25
good one

Reply
Michael Peter Gower (@MichaelPtrGower) says:
April 5, 2013 at 16:48
Informative and helpful article with great examples (and I just read an O'Reilly book on this!) I've linked to you from my blog, Hacker's Valhalla (http://hackersv.blogspot.com/2013/04/senior-developer-interviews-tales-from.html). As an example of needing to understand JavaScript prototyping.

Reply
Kite Flying Game says:
April 9, 2013 at 04:32
its a nice tutorial on prototype in JavaScript. it helps me a lot.
thanks

Reply
Pingback: JavaScript Object Creation Patterns | Binarymist
Pingback: Steve Bishop's Blog | Old Bookmarks
Pingback: Javascript Prototypes
sakiski says:
August 21, 2013 at 11:43
This article doesn't even try to explain prototype. It even said every JS object has a prototype. What a complete bs.

Reply
Pingback: Closure & Prototype ? / Nobe4 Blog
Pingback: Good To Know: ASP .NET MVC Reference Guide | For the love of problems :)
Pingback: Contents and Topics Covered in Microsoft Exam 70-480: Programming in HTML5 with JavaScript and CSS3 Part 1 | Developing with your head in the clouds
Anonymous says:
April 6, 2016 at 12:07
It should be more clearly stated in the beginning of the text how the structure of a construction function and an object looks like, e.g. an object has a.__proto__ whereas the function A

Reply