Title: Understanding Javascript Prototypes

Answer:

creating objects:

Example A

```
var catA = {name: "Fluffy", color: "White", age: 0};
```

Example B

```
var catB = Object.create(new Object());
catB.name = "Fluffy";
catB.color = "White";
catB.age = 0;
```

Example C

```
function Cat(name, color) {
  this.name = name;
  this.color = color;
}
Cat.prototype.age = 0;

var catC = new Cat("Fluffy", "White");
```

In these examples, A and B produce identical results while example C produces a something different. Example B is not very commonly used since Example A is much less verbose and equally useful. While example A is very common and quite useful, it doesn't apply much to this blog post so I will be almost solely working with example C, which uses `function Cat` , for the remainder of this post.

Why is Cat a function instead of a class?

A constructor function really is no different than any other function, in fact the term "constructor function" is just a common terminology that suggests that the function will create properties on the object being created (represented by *this*). In other words, when we say `var catC = new Cat("Fluffy", "White")` , all that really is happening is the catC object is created and assigned to *this* for the duration of the call to the Cat function. Therefore, inside the function, `this.name = name;` is the same as saying `catC.name = name` . Hopefully, that helps to clarify that constructor functions are not doing anything magic, they are just normal functions that are creating properties on *this* -- even though we use them in a way that is similar to how classes work in other languages. In fact, using the *new* keyword, you can create an object out of any function (it just doesn't make sense with other functions).

Cat.prototype.age? What's that all about?

Ok, first of all, notice that `Cat.prototype.age` isn't really part of the Cat function. It's also helpful to realize when this code gets executed. Seeing the code all lumped together like that might fool you into thinking that `Cat.prototype.age = 0;` is being called when a new cat is created, which of course, on closer inspection is obviously not true. Usually this code (including the creation of the Cat function above it) is called when a web page is first loaded; thus making the Cat function available to be called. So think about what is happening there. The Cat function is being created and then the age property is being created (with a value of 0) on the Cat function's prototype.

Wait, functions have a prototype? What's a prototype?

Functions can be used to create class-like functionality in JavaScript; and all functions have a prototype property. That prototype property is somewhat like a class definition in other object-oriented langauge; but it is more than that. It is actually an instance of an object and every function in JavaScript has one whether you use it or not. Every function (constructor function or not) has a prototype object instance hanging off of it, interesting, huh?

So, let's talk about how this prototype object is actually used. When you create an object using the *new* keyword, it creates a new object, passes it in as *this* to the constructor function (letting that function do to it whatever it wants) and then, and this is the important part, it takes the object instance that is pointed to by that function's prototype property and assigns it as the prototype for that object.

So objects have a prototype property too?

Well, yes but you don't access it the same way and it isn't used for the same purpose as a function's prototype. And this is where a lot of confusion usually begins with prototypes (don't worry if your confusion began long before this, that's normal too). Objects do not have a *prototype* property, but they do have a prototype. Only, the word prototype means something different when talking about objects than it does when talking about functions. It would be nice if we had a different word, for now, let's call it proto (you'll understand why in a minute). So functions have prototypes and objects have protos. They are very similar, in fact, a function's prototype and an object's proto often (in fact, usually) point to the same object in memory.

An object's proto is the object from which it is inheriting properties. Notice I said it is the *object* from which it is inheriting properties, not the *function* or *class*, it really is an object instance in memory. This differs from a function's prototype which is used as the object to be assigned as the proto for new objects' created using this function as a constructor function. An object's proto can be retrieved by calling myObject.__proto__ in most browsers (thanks IE) and by calling Object.getPrototypeOf(myObject) in all browsers -- I'll use __proto__ going forward for simplicity.

So let's give an actual definition to each of these terms:

- **A function's prototype:** A function's prototype is the *object instance* that will become the prototype for all objects created using this function as a constructor.
- **An object's prototype:** An object prototype is the *object instance* from which the object is inherited.

Back to the examples

All that explanation can be confusing without showing some examples, so let's dive in. If you inspect the prototype of the cats in Examples A and B you get the following result:

```
catA.__proto__;
Object { }
```

But, if you inspect the cat in Example C you get this result:

```
catC.__proto__;
Cat {age: 0}
```

Side note: I am using chrome's developer console to execute these statements and I will be displaying the results shown by chrome in grey text as shown above.

Note that in examples A and B, they do have a prototype (__proto__) even though they were not created using a constructor function. Since we didn't specify a prototype for these, they have a prototype of Object. It is possible to create objects without a prototype using this syntax: `var myObject = Object.create(null);` , but that is very rare and I've never seen a use for it. So barring that example, I think it is safe to say that all objects eventually inherit from Object.

This is even true of catC; look what happens when we inspect catC further:

```
catC.__proto__;
Cat {age: 0}

catC.__proto__.__proto__;
Object { }

catC.__proto__.__proto__.__proto__;
null
```

Notice that catC has a prototype (__proto__) of Cat. Actually, to say it that way is not really accurate. Cat is a function, so it cannot be a prototype; remember in the definitions above that a prototype is not a function it is an *object instance*. This is important to remember when thinking about prototypes -- an object's prototype is not the function that created it but an instance of an object. Since this is important, let's explore it a bit further:

```
Cat;
function Cat(name, color) {
  this.name = name;
  this.color = color;
}

Cat.prototype;
Cat {age: 0}

catC;
Cat {name: "Fluffy", color: "White", age: 0}
```

Look at the difference between Cat, Cat.prototype and catC. What this is showing is that Cat is a function but Cat.prototype and catC are objects. It further shows that Cat.prototype has an age property (set to 0) and catC has three properties -- including age...sort of (stay tuned). When you define a function, it creates more than just the function, it also creates a new *object* with that function as it's type and assigns that new *object* to the function's prototype property. When we first created the Cat function, before we executed the line `Cat.prototype.age = 0;`, if we would have inspected Cat's prototype it would have looked like this: `Cat {}`, an object with no properties, but of type Cat. It was only after we called `Cat.prototype.age = 0;` that it looked like this: `Cat {age: 0}`.

Inherited properties vs. native properties

In the above paragraph, I stated that catC had "three properties -- including age...sort of (stay tuned)"...thanks for sticking with me. Age really isn't a direct property of catC. You can see this by executing these statements:

```
catC.hasOwnProperty("name");
true

catC.hasOwnProperty("color");
true

catC.hasOwnProperty("age");
false
```

This is because age, actually belongs to catC's prototype; and yet, if I execute the statement `catC.age;`, it does indeed return `0`. What is actually happening here, is when we ask for `catC.age`, it checks to see if catC has a property named age, and if it does it returns it, if not, it asks catC's prototype if it has an age property. It continues doing this all the way up the prototype chain until it finds the matching property or finds an object with a null prototype and if it doesn't find the property in the prototype chain it will return undefined.

So that's what prototype chaining is?

Yep. You may have heard of prototype chaining before. It is really quite simple to understand now that you (hopefully) understand a little more about how prototypes work. A prototype chain is basically a linked-list of objects pointing backwards to the object from which each one inherits.

Changing an function's prototype

Remember that a function's prototype is just an object, so what would happen if we started changing the properties of a function's prototype after we created objects from it? Consider the following examples:

```
function Cat(name, color) {
  this.name = name;
  this.color = color;
}
Cat.prototype.age = 3;

var fluffy = new Cat("Fluffy", "White");
var scratchy = new Cat("Scratchy", "Black");

fluffy.age;
3

scratchy.age;
3

Cat.prototype.age = 4;

fluffy.age;
4

scratchy.age;
4
```

So, notice that changing the age of the Cat function's prototype property also changed the age of the cats that had inherited from it. This is because when the Cat function was created so was it's prototype object; and every object that inherited from it inherited this instance of the prototype object as their prototype. Consider the following example:

```
function Cat(name, color) {
  this.name = name;
  this.color = color;
}
Cat.prototype.age = 3;

var fluffy = new Cat("Fluffy", "White");
var scratchy = new Cat("Scratchy", "Black");

fluffy.age;
3

scratchy.age;
3

Cat.prototype = {age: 4};

fluffy.age;
3

scratchy.age;
3

var muffin = new Cat("Muffin", "Brown");

muffin.age;
4
```

First, notice that I did not just change the value of the prototype.age property to 4, I actually changed the Cat function's prototype to point to a new object. So while Muffin inherited the new prototype object, Fluffy's and Scratchy's prototypes are still pointing to their original prototype object, which they originally inherited from the Cat function. This illustrates the point that a function's prototype property "is the *object instance* which will become the prototype (or __proto__) for objects created using this function as a constructor."

One more example to illustrate this point. What would happen if I changed the value of the age property of fluffy's prototype? Would this be different than simply changing fluffy's age? Yes, it would be different. Think about the examples above, and about how age is not actually a property on fluffy, it is a property of fluffy's prototype.

So, given this setup:

```
function Cat(name, color) {
  this.name = name;
  this.color = color;
}
Cat.prototype.age = 3;

var fluffy = new Cat("Fluffy", "White");
var scratchy = new Cat("Scratchy", "Black");
```

Compare this example:

```
fluffy.age = 4;

fluffy.age;
4

scratchy.age;
3
```

To this example:

```
fluffy.__proto__.age = 4;

fluffy.age;
4

scratchy.age;
4
```

These produce different results because in the first example, we are just adding a new age property to the fluffy. So in the first example both fluffy and fluffy.__proto__ have age properties with the vales 4 and 3, respectively. When you ask for fluffy.age, it finds the property on the fluffy object so returns it immediately without ever looking up the prototype chain.

Whereas in the second example, fluffy still does not have an age property, but it's prototype (which is the same instance in memory as scratchy's prototype) now has a the new value 4 thus affecting both fluffy's and scratchy's ages.

Multiple Levels of Inheritance

I have to confess, this is not something I use often because I tend to prefer composition over inheritance; but that is not to say it is without a place. Say you want to create a constructor function that "inherits" from another constructor function, much like you would do in other languages when you create one class that inherits from another. Let's look at an example of how you may do this:

```
function Animal(name) {
  this.name = name;
}
Animal.prototype.age=1;

function Cat(name, color) {
  Animal.call(this, name);
  this.color = color;
}
Cat.prototype = new Animal(null);

var catC = new Cat("Fluffy", "White");

catC.name;
Fluffy

catC.color;
White

catC.age;
1

catC.hasOwnProperty("name");
true

catC.hasOwnProperty("color");
true

catC.hasOwnProperty("age");
false
```

Notice that age is the only property that is not a direct property of catC. This is because when we called the Cat constructor function, it passed in our new object (catC/this) to the Animal function which created the name property on the object. The Cat function then also added the color property to catC/this. But the age property was added to the Animal function's prototype, it was never added directly to catC.

Inheriting Functions

In all the examples above, I used properties like name, color, and age to illustrate objects and inheritance. However, everything that I have done above with properties can be done with functions. If we were to create a speak() function on the Cat Function like this: `Cat.prototype.speak = function() { alert('meow'); };` , that function would be inherited by all objects that have Cat as their prototype, just like with the name, color and age properties.

Tags: functions / methods