justjs: node.js tutorials

New here? You might want to start at the beginning.

"this" considered harmful (sometimes)

4/19 '14

Object oriented programming in JavaScript can be complicated, with pitfalls and sneaky details to remember. And while certain aspects usually stay the same, there are as many styles of JavaScript OOP as there are JavaScript frameworks, because there is no official notion of a "class" in JavaScript.

Most JavaScript implementations of "classes" do have certain features in common. They rely on the "this" keyword to refer to the current object; after all, it's built into the language. They provide a convenience function to implement subclassing, because it's tricky to get right, especially in older browsers. And they have a really tough time handling callbacks in methods, because "this" ceases to refer to the object you expect.

But there is an alternative pattern that is surprisingly popular, in a quiet way. Developers tend to discover it independently. I think it has many benefits, and while it may not be perfect for every project, there's a strong case to be made for it in many. I like to call it the "self pattern."

Here's what a simple class looks like with the self pattern:

```
function Dog(name) {
  var self = this;
  self.name = name;

  self.bark = function() {
    console.log(self.name + ' barked.');
  };
}

var dog = new Dog('spot');
dog.bark();
```

All the methods are tucked right inside the constructor, which begins with "var self = this". And methods always refer to self... never this. Now we have a safe, lexically scoped way to refer to the object we really want.

Subclassing is quite easy with this pattern, and there's no need for a convenience function. If I want to subclass Dog to make Doge, I just do this:

```
function Doge(name) {
  var self = this;
  Dog.call(self, name);

  self.meme = function() {
    console.log('such oop very code');
  };
}
```

```
var doge = new Doge('doge');
doge.meme();
doge.bark();
```

In addition to "var self = this", a subclass must invoke the parent class constructor with "call" and any additional arguments. Not much overhead, considering you only have to write one constructor per class.

The self pattern also makes it easy to call the original version of a method that I'm redefining in a subclass. If I want to take advantage of the existing "bark" method while still extending its capabilities, I do this:

```
function Doge(name) {
  var self = this;
  Dog.call(self, name);

  var superBark = self.bark;
  self.bark = function() {
    console.log('many');
    superBark();
  };
}

var doge = new Doge('doge');
doge.bark();
```

And If I want to invoke a method as a callback, I can just pass it like any other function:

```
var doge = new Doge('doge');
setTimeout(doge.meme, 1000);
```

Advantages of the "self" pattern

This simple approach has the following advantages over prototypal inheritance:

1. **Subclassing is simple and straightforward.** JavaScript has no "extend" keyword out of the box, just a pile of tinker toys from which it might be built. Every JavaScript framework has its own implementation of subclassing, working around JavaScript's design flaws. If you don't mind locking out 12-20% of all web users (IE8), you can use a solution based on Object.create like "util.inherits" in node, but it's still a lot of code just to get what class-based languages give you for free, and it's different for each framework.

2. **Calling the original version of a method you're overriding is simple and straightforward.** In plain vanilla prototypal inheritance, writing your own constructors, it looks like:

```
Dog.bark.call(this);
```

I have to play with "call" and "this" each time, passing the original arguments to bark (if any) after passing "this". It's a little goofy.

I also have to hardcode the superclass name (Dog) or do gymnastics to avoid it without polluting the global namespace.

With Backbone it looks like this:

```
MyModel.__super__.clone.call(this);
```

With the "self" pattern, we can just grab a reference to the "old" method before we install the new one, following what I refer to as the "var super pattern":

```
var superBark = self.bark; // Save the superclass version of bark
self.bark = function() {
  superBark(); // Just call the darn thing. It can already see "self"
  // Now do my own stuff
};
```

3. **"Private" properties and methods are a fringe benefit.** Want to have properties and methods that other code can't see? Just declare ordinary variables and functions in the constructor, without making them properties of "self." Now you can call them from your constructor and the other methods, but outsiders can't see them.

4. **Handling callbacks inside a method is simple and straightforward.** This is a big one. As javascript developers, we are constantly creating callbacks.

But when those callbacks are invoked, "this" no longer points to the object we had in mind.

Here's an example of how this goes wrong when we use the "this" keyword in our methods. Sensible-looking code like this just doesn't work:

```
// This doesn't work

function Dog(name) {
  this.name = name;
}

Dog.prototype.bark = function() {
  console.log(this.name + ' barked.');
};

Dog.prototype.barkAfterOneSecond = function() {
  setTimeout(function() {
    this.bark();
  }, 1000);
};

var dog = new Dog('sandy');

// Works fine
dog.bark();

// Fails miserably
dog.barkAfterOneSecond();
```

"this" will refer to the timer object, which has no "bark" method, and an error results.

So how do developers work around this? Basically, by adopting the "self pattern" after all... but doing it over and over, once for every method:

```
Dog.prototype.barkAfterOneSecond = function() {
  // This works... by following the "self" pattern after all
  var self = this;
  setTimeout(function() {
    self.bark();
  }, 1000);
};
```

Or they do it with convenience functions like underscore's "bind":

```
setTimeout(_.bind(this.bark, this), 100);
```

Either way it's extra work each time and creates a lot of potential for buggy code.

When all of your methods are created inside the constructor in the first place, so that they can always see "self", this bug never happens. You can just write:

```
setTimeout(self.bark, 1000);
```

... And because "bark" was defined inside a closure that defines "self" and never refers to "this," the "callback bug" never happens. In fact, you can pass a method directly as a callback function.

Memory and time: a valid concern

There is one area where the prototype keyword does beat the self pattern. It is true that creating objects in this way, with a closure to contain "self" and methods being added to each individual object, takes more memory and time than the prototypal version.

The flyweight pattern

First: do we care? How much this matters depends on your application. But if you're going to create tens of thousands of objects, I would recommend that you follow the well-known flyweight pattern, in which all of the instance objects are simple JSON-friendly objects without methods and a single "manager" object manipulates them.

The flyweight pattern looks like this:

```
function Dogs() {
  var self = this;
  self.bark = function(dog) {
    console.log(dog.name + ' has barked');
  };
}

var dogs = new Dogs();
var zillionsOfDogs = [ { name: 'bella' }, { name: 'tuuli' } ];
_.each(zillionsOfDogs, function(dog) {
  dogs.bark(dog);
});
```

Using simple JSON-friendly objects in this way has another benefit: you can pass them back and forth to browsers and NoSQL databases as plain JSON objects, because they are.

Subclassing and the Flyweight Pattern

Note that you can still take advantage of subclassing and code reuse. The heavyweight "manager" objects can still subclass each other. And if you have mixed breeds of dogs in your data, a "type" property of each dog can be used to decide which manager to call:

```
managers[dog.type].bark(dog);
```

Yes, the manager pattern loses some elegance over calling methods directly on the dog objects. But because it is

explicit — each method explicitly takes an object as its first parameter — it is still safer and less bug-prone than the counterintuitive behavior of "this" in callback-driven code.

Still, your mileage may vary. Perhaps you feel it's worth typing "var self = this" at the top of each method to combine the performance advantages of the prototype chain with sane callbacks.

In any case, you really don't need the flyweight pattern unless you're creating enough distinct objects for performance issues to come to the fore. And modern implementations of JavaScript are very fast and efficient to begin with, especially compared to other languages of comparable convenience like Python and Ruby. You may find it takes quite a lot of objects for this issue to become relevant.

Save Your Skills For Paying The Bills

In the end what matters to me is writing simple, intelligible, bug-free code. The "self pattern" helps me get there. In cases with many thousands of objects or more, I add the flyweight pattern to avoid performance issues and still code in a simple, gotcha-free style.

The less time I spend wrestling with my tools, the more I can spent wrestling with the problem at hand. Reducing the cognitive load imposed by the language itself is a great way to reduce the number of WTFs per minute.
