A Drip of JavaScript

What are Prototype Properties and Methods?

Originally published in the A Drip of JavaScript newsletter.

A couple of weeks ago, I mentioned prototype properties in passing. But what exactly are they? Let's take a look.

```
function Book(title, author) {
    this.title = title;
    this.author = author;
}

Book.prototype.getFormatted = function () {
    return this.title + " - " + this.author;
}

var hobbit = new Book("The Hobbit", "Tolkien");

// Outputs: "The Hobbit - Tolkien"
console.log(hobbit.getFormatted());
```

What is this magical `getFormatted`? And how did it get attached to my object? It's all through the power of prototypes.

Whenever you create an object using a constructor, that constructor has a property called `prototype` which points to an object. If you haven't done anything special,

that prototype property is just an empty object. But it's an empty object with superpowers.

Let's see how that works.

```
function Book(title, author) {
    this.title = title;
    this.author = author;
}

Book.prototype.pubYear = "unknown";

var hobbit = new Book("The Hobbit", "Tolkien");

var caspian = new Book("Prince Caspian", "Lewis");
caspian.pubYear = 1951;

// Outputs: "unknown"
console.log(hobbit.pubYear);

// Outputs: 1951
console.log(caspian.pubYear);

// Outputs: "unknown"
console.log(Book.prototype.pubYear);
```

In the example above, we create a `Book` constructor, and then create a `pubYear` property "unknown" on it's prototype object.

When we try to access `hobbit.pubYear`, the JavaScript interpreter realizes that the `hobbit` object doesn't have a matching property, so it then checks to see if there is a matching property on the prototype object. Since there is, it will give us the value of `Book.prototype.pubYear`.

But because the `caspian` object has a `pubYear` property of it's own, the interpreter never has to go look at the prototype object.

While ordinary properties on a prototype can be useful, methods are more useful still. Let's go back to our original example.

```
function Book(title, author) {
    this.title = title;
    this.author = author;
}

Book.prototype.getFormatted = function () {
    return this.title + " - " + this.author;
}

var hobbit = new Book("The Hobbit", "Tolkien");

// Outputs: "The Hobbit - Tolkien"
console.log(hobbit.getFormatted());
```

Now you should know how the property lookup works. The interpreter realizes that there is no `getFormatted` property on `hobbit` itself, so looks for it on the prototype.

But you might notice something else special. The `getFormatted` method makes use of `this.title` and `this.author`. That works because when an object's method is invoked, the object itself becomes the `this` for the method. And that's even if the method itself belongs to the prototype rather than directly to the object.

Why bother with all of this, though? Why not define `getFormatted` directly on the object, like so?

```
function Book(title, author) {
    this.title = title;
    this.author = author;

    this.getFormatted = function () {
        return this.title + " - " + this.author;
```

```
    }
}
```

Because if you do it like that, rather than defining a single method which can be used by all `Book` objects, each object has its own copy of the method. Now imagine a scenario where you are producing lots of `Book` objects.

```
var bookArray = [];

for (var i = 0; i < 100; i++) {
    bookArray[i] = new Book("Some Title", "Some Author");
}
```

Now you have have 100 copies of `getFormatted` taking up space in memory. In addition, if you need to change how `getFormatted` works, you'll need to manually update each instance of `Book`. Compare that to how simple it is to update the prototype method.

```
function Book(title, author) {
    this.title = title;
    this.author = author;
}

Book.prototype.getFormatted = function () {
    return this.title + " - " + this.author;
}

var hobbit = new Book("The Hobbit", "Tolkien");

// Outputs: "The Hobbit - Tolkien"
console.log(hobbit.getFormatted());

// Oops! We need to use commas, not hyphens
Book.prototype.getFormatted = function () {
    return this.title + ", " + this.author;
}
```

```
// Outputs: "The Hobbit, Tolkien"
console.log(hobbit.getFormatted());
```

Updating the method on the prototype means that all instances of `Book` get the updated method immediately, and you don't have the possibility of some instances having the outdated method.

So those are the basics of how prototype properties and methods work.

Whew! That's a bit wordier than normal, but the newsletter should be back to a normal length next week.

Thanks for reading!

Josh Clanton
