A Drip of JavaScript

# Basic Inheritance with JavaScript Constructors

*Originally published in the A Drip of JavaScript newsletter.*

We've looked before at using JavaScript's constructors to create our own custom object types. But what we didn't look at was how we can create an inheritance hierarchy.

It's important to note that even though constructors are often referred to as "classes," they really aren't the same thing as classes in other languages. In JavaScript, a constructor is just a function invoked by the `new` operator which builds a new object.

Here's a little refresher:

```
function SuperHuman (name, superPower) {
    this.name = name;
    this.superPower = superPower;
}

SuperHuman.prototype.usePower = function () {
    console.log(this.superPower + "!");
};

var banshee = new SuperHuman("Silver Banshee", "sonic wail");

// Outputs: "sonic wail!"
banshee.usePower();
```

The `SuperHuman` constructor contains our initialization logic, while `SuperHuman.prototype` contains the methods that are shared across all `SuperHuman` instances.

But suppose that we want to create a new type which inherits from `SuperHuman` while adding its own functionality? What would that look like?

```
function SuperHero (name, superPower) {
    this.name = name;
    this.superPower = superPower;
    this.allegiance = "Good";
}

SuperHero.prototype.saveTheDay = function () {
    console.log(this.name + " saved the day!");
};

var marvel = new SuperHero("Captain Marvel", "magic");

// Outputs: "Captain Marvel saved the day!"
marvel.saveTheDay();
```

While this gets us started, there are a couple of problems. First of all, the `SuperHero` constructor is repeating some of the logic of the `SuperHuman` constructor. And more importantly, at this point instances of `SuperHero` don't have access to `SuperHuman` methods. For example:

```
// TypeError: Object <#SuperHero> has no method 'usePower'
marvel.usePower();
```

Let's fix those couple of issues.

```
function SuperHero (name, superPower) {
    // Reuse SuperHuman initialization
    SuperHuman.call(this, name, superPower);

    this.allegiance = "Good";
}

SuperHero.prototype = new SuperHuman();

SuperHero.prototype.saveTheDay = function () {
    console.log(this.name + " saved the day!");
};

var marvel = new SuperHero("Captain Marvel", "magic");

// Outputs: "Captain Marvel saved the day!"
marvel.saveTheDay();

// Outputs: "magic!"
marvel.usePower();
```

We've managed to eliminate the repeated constructor logic by calling `SuperHuman` with `SuperHero`'s `this` object and passing along the necessary arguments. That ensures that `SuperHuman`'s initialization logic will act on the new `SuperHero` object. And then we tack on the additional logic that is specific to `SuperHero`.

But where the inheritance comes in is on `SuperHero.prototype`. In order to ensure that it inherits the methods from `SuperHuman.prototype`, we actually make it an instance of `SuperHuman` with `new SuperHuman()`.

This basic inheritance pattern won't always work, particularly if the parent constructor is complex, but it will handle simple situations quite well.

In future issues we'll take a look at more sophisticated ways of doing inheritance.

---