A Drip of JavaScript

Dealing with the Dangers of `this` in Constructors

Originally published in the A Drip of JavaScript newsletter.

As I mentioned in the last issue, invoking a constructor without `new` can be dangerous. Let's go over why.

```
function Color(r, g, b) {
    this.r = r;
    this.g = g;
    this.b = b;
}

// Safe invocation
var red = new Color(255, 0, 0);

// Dangerous invocation
var blue = Color(0, 0, 255);
```

When a constructor is invoked with the `new` keyword, the `this` value of the constructor is set to the new object that you are creating. But when a constructor is invoked like a normal function, its `this` defaults to the same `this` variable that any other function gets. And normally that is the global object (`window` in the browser.)

Here is an illustration of the problem.

```
// Global variable
r = "Rodent Of Unusual Size";

function Color(r, g, b) {
    this.r = r;
    this.g = g;
    this.b = b;
}

// Dangerous invocation
// Means `this` is the global object
var blue = Color(0, 0, 255);

// Outputs: 0
console.log(r);

// Outputs: undefined
console.log(blue);
```

In the example above, there is a global variable named `r`. Or to put it another way, the global object has a property named `r`. When the `Color` constructor is invoked without `new`, the constructor's `this` is set to the global object (in most cases). Which means that the constructor function has just overwritten the global `r` variable with something that was intended to be a property of the `blue` object.

Furthermore, because `Color` was invoked as an ordinary function, it didn't automatically return a new object, which means that `blue` is also undefined.

As you can imagine, debugging an issue like this can be time consuming and frustrating. So how do we prevent these sorts of problems? Fortunately the answer is pretty straightforward.

```
// Global variable
r = "Rodent Of Unusual Size";

function Color(r, g, b) {
```

```
    // Check whether `this` is a
    // `Color` object.
    if (this instanceof Color) {
        this.r = r;
        this.g = g;
        this.b = b;
    } else {
        // If not, then we should invoke
        // the constructor correctly.
        return new Color(r, g, b);
    }
}

// Dangerous invocation
// Means `this` is the global object
var blue = Color(0, 0, 255);

// Outputs: "Rodent Of Unusual Size"
console.log(r);

// Outputs: Color {r: 0, g: 0, b: 255}
console.log(blue);
```

In the updated `Color` constructor, the first thing we do is check whether `this` is an instance of `Color`. It works because the `new` keyword will have already created the new object as an instance of the constructor before the constructor function begins running.

If it isn't a `Color` object, then we know the constructor was invoked incorrectly, so we skip all the construction logic and have `Color` return the results of correctly invoking itself with `new`.

This means that the constructor is no longer in danger of clobbering the global object's properties.

Of course, using this approach also means that developers may get into the bad habit of invoking constructors without `new`. If you'd rather just force them to always use

`new`, you could throw an error instead, like so:

```
function Color(r, g, b) {
    // Check whether `this` is a
    // `Color` object.
    if (this instanceof Color) {
        this.r = r;
        this.g = g;
        this.b = b;
    } else {
        // If not, throw error.
        throw new Error("`Color` invoked without `new`");
    }
}
```

And that's how you can make your custom constructors safely deal with a missing `new` keyword.

Josh Clanton
