A Drip of JavaScript

JavaScript's Primitive Wrapper Objects

Originally published in the A Drip of JavaScript newsletter.

We've talked before about how in JavaScript most things behave like objects even when they aren't objects. For instance, consider how we can call methods on a string even though it is a primitive:

```
// Outputs: "FRED FLINTSTONE"
console.log("Fred Flintstone".toUpperCase());
```

How does that work, though? Initially you might think that strings are really objects in disguise and try assigning properties to them.

```
var fred = "Fred Flintstone";
fred.favoriteFood = "Brontosaurus Steak";

// Outputs: undefined
console.log(fred.favoriteFood);
```

But that doesn't work. And even more strangely, it doesn't trigger an error. It turns out that in order to allow you to call methods on a primitive, JavaScript does a little bit of trickery which we'll get to shortly.

Apart from `null` and `undefined`, each primitive type (string, number and boolean) has a corresponding object equivalent which you can create by invoking its

constructor. For instance:

```
var barney = new String("Barney Rubble");

// Outputs: "Barney Rubble"
console.log(barney);

barney.favoriteFood = "Pterodactyl Eggs";

// Outputs: "Pterodactyl Eggs"
console.log(barney.favoriteFood);

// Outputs: "object"
console.log(typeof barney);
```

As you can see, though, the string object can have properties assigned to it, and it reports itself to be of type "object."

The trickery I mentioned before is that any time you attempt to access a property on a primitive, JavaScript will implicitly create a temporary wrapper object. We can verify this by doing the following:

```
String.prototype.reportType = function () {
    return typeof this;
};

var fred = "Fred Flintstone";

// Outputs: "string"
console.log(typeof fred);

// Outputs: "object"
console.log(fred.reportType());
```

When we directly check the type of a string primitive we get `"string"` as expected, but when we check the type of `this` in a method executed on a string primitive we

get `"object"`.

The JavaScript engine doesn't keep this wrapper object around, though. As soon as the work of the method (or other property) is done, it is disposed of.

This explains why trying to assign properties to a primitive doesn't work, but also doesn't throw an error. Assigning the property succeeds, but the property is set on a wrapper object which is immediately destroyed. So when you go to look up the property later, there is nothing there anymore.

I hope this has helped you better learn the quirks of JavaScript primitives.
