A Drip of JavaScript

Numbers and JavaScript's Dot Notation

Originally published in the A Drip of JavaScript newsletter.

In JavaScript almost everything behaves like an object. And that is true even when the thing in question isn't an object. For instance:

```
var numOfIs = "Odin".match(/i/).length;

// Outputs: 1
console.log(numOfIs);

var strOfTrue = true.toString();

// Outputs: "true"
console.log(strOfTrue);
```

Here we see two examples. First, the primitive string `"Odin"` lets us treat it as if it were an object by accessing the `match` method. Second, the primitive boolean value `true` lets us treat it as if it were an object by accessing the `toString` method.

However, if we try to do the same thing with a number, we are likely to run into an error.

```
// SyntaxError: Unexpected token ILLEGAL
var meaningOfLife = 42.toString();
```

Because of this error, many beginning JavaScript developers get the impression that you can't access properties on a primitive number with dot notation. You might try bracket notation instead:

```
var meaningOfLife = 42["toString"]();

// Outputs: "42"
console.log(meaningOfLife);
```

That works, but it's not especially nice. And it turns out that dot notation does work with **some** numbers.

```
var platform = 9.75.toString();

// Outputs: "9.75"
console.log(platform);
```

Why does dot notation work with `9.75` but not with `42`? The secret is in the dots. When a JavaScript engine parses your source code, it has to interpret just what a dot means in a given context.

In the case of `42.toString()` the dot is potentially ambiguous. Does it mean the decimal separator? Or does it mean object member access? JavaScript opts to interpret all integers followed by a dot as representing part of a floating point number. But as there is no such number `42.toString()`, it raises a SyntaxError.

In the case of `9.75.toString()` the first dot was unambiguously part of a floating point number, and the second dot was unambiguously object member access.

So the way to avoid problems with integers like `42` is to make sure that the parser can't mistake your dot notation for a decimal point. There are at least three ways to do that.

```
var meaningOfLife = 42..toString();
```

The double-dot approach above is syntactically equivalent to typing `42.0.toString()`. And since JavaScript doesn't have separate types for integers and floating point numbers, the internal value of the number is the same as well.

```
var meaningOfLife = 42 .toString();
```

This one is confusing at first glance. There is only one dot here, so how is it working as member access? The difference is the space between `42` and the the dot. JavaScript numbers cannot have spaces between the integer portion and the decimal point portion. Because of this, the dot is interpreted as member access. However, because of its poor readability and the tendency other developers will have to "fix" it, this is a bad choice for production code.

```
var meaningOfLife = (42).toString();
```

Having looked at the examples above, you'll understand why this one works. The parentheses ensure that the parser doesn't get confused. This is also by far my favorite approach for use in production code. It is highly readable, and the use of parentheses clearly communicates that `toString` is being called on the value of `42`.

That's it for this week. Thanks for reading!

Josh Clanton
