# Creating Unwritable Properties with Object.defineProperty

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

JavaScript has a reputation as a language where the developer can redefine just about anything. While that has been largely true in past versions of JavaScript, with ECMAScript 5 the situation has begun to change. For instance, thanks to `Object.defineProperty` it is now possible to create object properties that cannot be modified.

Why would you want to do that? Imagine that you are building a mathematics library.

```
var principia = {
    constants: {
        // Pi is used here for convenience of illustration.
        // For real applications, you'd want to use Math.PI
        pi: 3.14
    },
    areaOfCircle: function (radius) {
        return this.constants.pi * radius * radius;
    }
};
```

Your library provides a collection of common equations, but for convenience also defines a set of mathematical constants. Of course, the thing about constants is that they should remain constant and shouldn't be redefined. But suppose a user of your library accidentally does something like this:

```
// Accidental assignment
if (principia.constants.pi = myval) {
    // do something
}
```

The user has accidentally assigned a value to `pi`. Suddenly every equation in your library that depends on `pi` will return incorrect results. This will likely lead the user to believe your library is defective rather than discovering the error in their code. While there are multiple ways to avoid this issue, the simplest solution is just to make `pi` unwritable. Let's take a look at how to do that.

```
var principia = {
    constants: {},
    areaOfCircle: function (radius) {
        return this.constants.pi * radius * radius;
    }
};

Object.defineProperty(principia.constants, "pi", {
    value: 3.14,
    writable: false
});
```

The `defineProperty` method takes three arguments: the object whose property you are creating or modifying, the name of the property, and an options object. The options object itself allows us to set several options, but for the moment we are only interested in `value` and `writable`.

The `value` option defines what the actual value of our property will be, while `writable` specifies whether we can assign a new value to it. So let's see how it works.

```
// Try to assign a new value to pi
principia.constants.pi = 2;
```

```
// Outputs: 3.14
console.log(principia.constants.pi);
```

As you can see, attempting to assign a new value to `pi` fails silently and `pi` retains its original value. So now a mistake by a careless library user won't cause problems with the library itself.

And if the user of your library is in strict mode, it gets better.

```
// Turn on strict mode
"use strict";

// TypeError: Cannot assign to read only property 'pi' of principia.cons
principia.constants.pi = 2;
```

Strict mode will throw an error if you attempt to assign a new value to unwritable properties, which means that errors like the one illustrated above would get flagged immediately.

There are two more aspects of `defineProperty` that you'll need to understand to use it effectively.

While in our example above, we explicity set `writable`, it turns out that `writable` defaults to false, so our earlier example can be rewritten like so.

```
Object.defineProperty(principia.constants, "pi", {
    value: 3.14
});
```

In addition there is another option called `configurable` which specifies whether you can use `defineProperty` and similar methods to reconfigure things like `writable` to a different state. The `configurable` option defaults to `false`, so if you want to let users of your library override `pi` intentionally, but not through accidental assignment, you would need to explicitly set it to `true`, like so:

```
Object.defineProperty(principia.constants, "pi", {
    value: 3.14,
    configurable: true
});
```

There is one rather large "gotcha" to be aware of when using `defineProperty`. While setting `writable` prevents assignment to the property, it does not make the property's value immutable. Consider the case of an array:

```
var container = {};

Object.defineProperty(container, "arr", {
    writable: false,
    value: ["a", "b"]
});

container.arr = ["new array"];

// Outputs: ["a", "b"]
console.log(container.arr);

container.arr.push("new value");

// Outputs: ["a", "b", "new value"]
console.log(container.arr);
```

The `arr` property is unwritable, so it will always point to the same array. But the array itself may be changed. Unless you use a value that is intrinsically immutable (like a string primitive), the property's value is still subject to change. We'll consider a way of locking down arrays and other objects in a future issue.

Unfortunately, because `Object.defineProperty` is part of ES5, it is only fully supported in IE9 and newer. IE8 has a partial implementation which only works on DOM objects, and would be useless for the examples considered above. Even more

unfortunately, there is no compatibility shim for IE8. However, if you don't need to deal with older browsers, `defineProperty` might be just what you are looking for.

Thanks for reading!

Joshua Clanton

---