A Drip of JavaScript

The Perils of Non-Local Mutation

Originally published in the A Drip of JavaScript newsletter.

One of the most important things to understand about JavaScript is how to deal with the mutable nature of objects. Unlike primitives (strings, numbers, booleans, etc.) objects are subject to modification.

Consider this simple example:

```
var solarSystem = {
    planets: 9,
    inhabited: "Earth"
};

var homeSystem = solarSystem;

// Scientists reclassify Pluto
homeSystem.planets = 8;

console.log(solarSystem);
// => { planets: 8, inhabited: "Earth" }
```

In this example, both `solarSystem` and `homeSystem` point to the same underlying object. And when the number of planets is modified, that is reflected in both variables.

Ordinarily there isn't much point to explicitly declaring two variables for the same object. However, it is incredibly common to do so **implicitly** when defining a function.

```
var solarSystem = {
    planets: 9,
    inhabited: "Earth"
};

function reclassifyPluto (system) {
    system.planets = 8;
}

reclassifyPluto(solarSystem);

console.log(solarSystem);
// => { planets: 8, inhabited: "Earth" }
```

Here `system` becomes a reference to `solarSystem`, which the `reclassifyPluto` function modifies directly. That may not seem like such a big deal. But it can have far-reaching consequences.

```
var solarSystem = {
    planets: 9,
    inhabited: "Earth"
};

function simulateMarsSettlement (system) {
    system.inhabited = "Earth & Mars";

    return system;
}

function reclassifyPluto (system) {
    system.planets = 8;
}
```

```
var simulatedSystem = simulateMarsSettlement(solarSystem);

console.log(simulatedSystem);
// => { planets: 9, inhabited: "Earth & Mars" }

reclassifyPluto(solarSystem);

console.log(solarSystem);
// => { planets: 8, inhabited: "Earth & Mars" }
```

Why does `solarSystem` think that both Earth and Mars are inhabited? Because `simulateMarsSettlement` directly modified it. Imagine how difficult it would be to debug if there were dozens of functions depending on the `solarSystem` object which suddenly changed.

We could code defensively by freezing `solarSystem` so that it can't be modified. But this would only solve part of our problem. The underlying issue is that the functions are modifying an object which doesn't "belong" to them.

In general it is better to avoid modifying objects which are passed in as function arguments, because you may not know what else is depending on them. If you need a modified version of argument, then creating a copy is probably your best solution.

```
var solarSystem = {
    planets: 9,
    inhabited: "Earth"
};

function simulateMarsSettlement (system) {
    var simulation = {
        planets: system.planets,
        inhabited: system.inhabited
    };

    simulation.inhabited = "Earth & Mars";

    return simulation;
```

```
}

function reclassifyPluto (system) {
    var reclassified = {
        planets: system.planets,
        inhabited: system.inhabited
    };

    reclassified.planets = 8;

    return reclassified;
}

var simulatedSystem = simulateMarsSettlement(solarSystem);

console.log(simulatedSystem);
// => { planets: 9, inhabited: "Earth & Mars" }

// Replace solarSystem with updated version
var solarSystem = reclassifyPluto(solarSystem);

console.log(solarSystem);
// => { planets: 8, inhabited: "Earth" }
```

Note that in this solution we do end up modifying the `solarSystem` variable. But we do it "locally," not hidden away inside a function. This keeps our code simple and easy to reason about.

Thanks for reading!

Josh Clanton
