A Drip of JavaScript

Negating Predicate Functions in JavaScript

Originally published in the A Drip of JavaScript newsletter.

While you may not have heard the term, chances are you've used predicate functions before. A predicate is essentially a function that determines whether something is `true` or `false` based on its arguments. It is common (though not necessary) for predicates to be named "isX", such as `isEven` or `isNumber`.

Suppose that we have a program which deals with cataloging comic book heroes and villains represented as simple objects:

```
var superman = {
    name: "Superman",
    strength: "Super",
    heroism: true
};
```

And as part of that program we have a number of predicates that might look something like this:

```
function isSuperStrong (character) {
    return character.strength === "Super";
}

function isNotSuperStrong (character) {
    return character.strength !== "Super";
```

```
}

function isHeroic (character) {
    return character.heroism === true;
}

function isNotHeroic (character) {
    return character.heroism !== true;
}

// Outputs: false
console.log(isNotSuperStrong(superman));

// Outputs: false
console.log(isNotHeroic(superman));
```

As you can see, this is a bit repetitive. But the problem isn't that the code is longer. Rather, the problem is that for each pair of predicates (the "is" and "isNot") we are defining our core logic twice. Having that logic repeated means we are more likely to make mistakes like updating only one of the predicates when our logic changes.

What can we do to fix that problem? Our first thought might be to do something like this:

```
function isSuperStrong (character) {
    return character.strength === "Super";
}

function isNotSuperStrong (character) {
    return !isSuperStrong(character);
}

function isHeroic (character) {
    return character.heroism === true;
}

function isNotHeroic (character) {
```

```
    return !isHeroic(character);
}

// Outputs: false
console.log(isNotSuperStrong(superman));

// Outputs: false
console.log(isNotHeroic(superman));
```

While this is certainly an improvement, we still have repetition of a different kind. Both of our "isNot" predicates share a piece of core logic. They both reverse the sense of the predicates that they are based upon.

Wouldn't it be nice if we could abstract that away into something clearer and more maintainable? Fortunately, we can.

```
function negate (predicateFunc) {
    return function () {
        return !predicateFunc.apply(this, arguments);
    };
}
```

This is another example of treating functions as first-class values in JavaScript. The `negate` function accepts a predicate as an argument, and returns a function whose sense is the opposite of the original predicate.

(If the usage of `apply` is confusing, you might want to read the previous drip on call and apply.)

Let's use `negate` on our original problem.

```
function isSuperStrong (character) {
    return character.strength === "Super";
}

var isNotSuperStrong = negate(isSuperStrong);
```

```
function isHeroic (character) {
    return character.heroism === true;
}

var isNotHeroic = negate(isHeroic);

// Outputs: false
console.log(isNotSuperStrong(superman));

// Outputs: false
console.log(isNotHeroic(superman));
```

Everything continues to work as expected. But now we've pulled the shared logic of the "isNot" predicates into one location. And it is much easier to tell at a glance that the definitions of the "isNot" predicates are derived from the "is" predicates.

Applied to only a couple of functions, this refactoring may not look like much. But applied to dozens of functions scattered throughout a complex system, `negate` can help to make things much more maintainable.

Both Underscore and Lo-Dash will include `negate` in future versions. It is also available right now as part of Underscore-contrib under the name `complement`.

Thanks for reading!

Josh Clanton
