Joel Thoms Follow
Computer Scientist and Technology Evangelist with 21 years of experience with JavaScript!
Jan 11 · 5 min read

Rethinking JavaScript: Death of the For Loop

JavaScript's `for loop` has served us well, but it is now obsolete and should be retired in favor of newer functional programming techniques.

Fortunately, this is a change that doesn't require you to be a functional programming master. Even better, **this is something you can do in existing projects today!**

What is the problem with JavaScript's for loop anyway?

The `for loop`'s design encourages **mutation of state** and usage of **side effects**, both of which are potential sources of buggy and unpredictable code.

We have all heard that global state is bad and should be avoided. Though, **local state shares the same evils as global state**, we just don't notice it as often because it's on a smaller scale. So we never actually solved the problem, we just minimized it.

With a mutable state, at some unknown point in time, a variable will change for an unknown reason and you will spend hours debugging and searching for the reason that value changed. I have already pulled a dozen hairs out of my head just thinking about this.

Next, I would like to quickly talk about **side effects**. Those words just sound terrible, side effects. Yuck. Do you want your program to have side effects? No, I don't want my programs to have side effects!

But what is a side effect?

A function is considered to have side effects is when it modifies something outside of the function's scope. It could be changing the value of a variable, reading keyboard input, making an api call, writing data to disk, logging to a console, etc.

Side effects are powerful, but with great power comes great responsibility.

Side effects should be eliminated when possible or encapsulated and managed. **Functions with side effects are harder to test and harder to reason about**, so get them out whenever you can. Fortunately we aren't going to worry about side effects here.

Okay, less words more code. Let's take a look at a typical `for loop` that you've probably seen a thousand times.

```
const cats = [
  { name: 'Mojo',    months: 84 },
  { name: 'Mao-Mao', months: 34 },
  { name: 'Waffles', months: 4 },
  { name: 'Pickles', months: 6 }
]


var kittens = []
```

```
// typical poorly written `for loop`
for (var i = 0; i < cats.length; i++) {
  if (cats[i].months < 7) {
    kittens.push(cats[i].name)
  }
}
```

```
console.log(kittens)
```

My plan is to refactor this code step by step so you can see how easy it is to transform your own code into something more beautiful.

The first change I want to make is to extract the `if` statement into it's own function.

```
const isKitten = cat => cat.months < 7

var kittens = []

for (var i = 0; i < cats.length; i++) {
  if (isKitten(cats[i])) {
    kittens.push(cats[i].name)
  }
}
```

It is a good practice in general to extract your `if` statements. **The change in filtering from "less than 7 months" to "is a kitten" is a big deal.** Now when you read the code the intent becomes clear. Why are we getting cats under 7 months? This is not clear at all. Our intent is to find kittens, so let the code say that!

Another benefit is `isKitten` is now reusable and we all know,

> *making our code reusable should always be one of our goals.*

The next change is to extract the transformation (or mapping) from an object of type cat to a name. This change will make more sense later, so for now you'll just have to trust me.

```
const isKitten = cat => cat.months < 7
const getName = cat => cat.name

var kittens = []

for (var i = 0; i < cats.length; i++) {
  if (isKitten(cats[i])) {
```

```
      kittens.push(getName(cats[i]))
    }
  }
```

I contemplated writing a few paragraphs to describe the mechanics of `filter` and `map`. But I think by not describing them and instead showing you how easily you can read and understand this code, even without ever having been introduced to `map` or `filter`, would best demonstrate how readable your code can become.

```
const isKitten = cat => cat.months < 7
const getName = cat => cat.name
```

```
const kittens =
  cats.filter(isKitten)
      .map(getName)
```

Also notice that we have eliminated `kittens.push(...)`. No more mutation of state and no more `var`!

> *Code that uses const (over var and let) is sexy as hell*

Full disclosure here, we could have used const the whole time because const doesn't make the object itself immutable (more on this another time). But hey, it's a contrived example, so cut me some slack!

The last refactoring I would suggest is to also extract the filtering and mapping to it's own function (you know, for that whole reuse thing).

And all together now:

```
const isKitten = cat => cat.months < 7
const getName = cat => cat.name
const getKittenNames = cats =>
  cats.filter(isKitten)
      .map(getName)
```

```
const cats = [
  { name: 'Mojo',    months: 84 },
  { name: 'Mao-Mao', months: 34 },
```

```
    { name: 'Waffles', months: 4 },
    { name: 'Pickles', months: 6 }
  ]


  const kittens = getKittenNames(cats)


  console.log(kittens)
```

For extra credit, how would you further decompose these functions? Think about the ‘less than’ or property ‘name’, ‘map’ or ‘filter’. And double extra credit you could research function composition.

Cheers!