Tip 8: Avoid Push Mutations with the Spread Operator

In this tip, you'll learn how to avoid array mutations by creating new arrays with the spread operator.

Avoiding mutations

As you've just seen, mutations can have unexpected consequences. If you change something in a collection early in the code, you can create a bug much deeper. Mutations may not always cause major headaches, but they do have that potential, so it's best to avoid them when possible. In fact, some popular Java- Script libraries (such as Redux) won't allow functions with any mutations at all.

Plus, a lot of modern JavaScript is functional in style, meaning you'll need to write code that doesn't contain side effects and mutations. There's a lot to be said about functional JavaScript, more than what can fit in this course. If you're interested, you can learn more in Functional JavaScript by Michael Fogus.

`push` method

I hope by now you understand why mutations are bad. But if you're like me, you probably wonder what does it all mean in practice? Consider a common array mutation: `push()`. The `push()` method changes the original array by adding an item to the end. When you add an item, you're mutating the original array. Fortunately, you can avoid the side effect with the spread operator.

Problem caused by using `push`

Before that, start with a problem caused by the `push()` method.

Imagine a simple function that takes a shopping cart and summarizes the contents. The function checks to see if there are too many discounts and returns an *error* object if there are. Otherwise, if the cart has enough items, it adds a free gift.

```
const cart = [
  {
    name: 'The Foundation Triology',
    price: 19.99,
    discount: false,
  },
  {
    name: 'Godel, Escher, Bach',
    price: 15.99,
    discount: false,
  },
  {
    name: 'Red Mars',
    price: 5.99,
    discount: true,
  },
];

const reward = {
  name: 'Guide to Science Fiction',
  discount: true,
  price: 0,
};

function addFreeGift(cart) {
  if (cart.length > 2) {
    cart.push(reward);
    return cart;
  }
  return cart;
}

function summarizeCart(cart) {
  const discountable = cart.filter(item => item.discount);
  if (discountable.length > 1) {
    return {
      error: 'Can only have one discount',
    };
  }
  const cartWithReward = addFreeGift(cart);
  return {
    discounts: discountable.length,
    items: cartWithReward.length,
    cart: cartWithReward,
  };
}


console.log("Add free gift:");
console.log(addFreeGift(cart));
console.log("\n");
console.log("Summarize cart:");
console.log(summarizeCart(cart));
```

The `cart` is a simple array, and the gift is merely an added item. The problem is this code is one line away from causing an error. As usual, take a moment and see if you can locate the problem.

This is a great example of why a mutation can seem so harmless. What if six months down the road, a well-meaning developer decides to clear things up by putting all the variable declarations at the top of the function?

```
const cart = [
  {
    name: 'The Foundation Triology',
    price: 19.99,
    discount: false,
  },
  {
    name: 'Godel, Escher, Bach',
    price: 15.99,
    discount: false,
  },
  {
    name: 'Red Mars',
    price: 5.99,
    discount: true,
  },
];

const reward = {
  name: 'Guide to Science Fiction',
  discount: true,
  price: 0,
};

function addFreeGift(cart) {
  if (cart.length > 2) {
    cart.push(reward);
    return cart;
  }
  return cart;
}

function summarizeCartUpdated(cart) {
  const cartWithReward = addFreeGift(cart);
  const discountable = cart.filter(item => item.discount);
  if (discountable.length > 1) {
    return {
      error: 'Can only have one discount',
    };
  }
  return {
    discounts: discountable.length,
    items: cartWithReward.length,
    cart: cartWithReward,
  };
}
```

```
console.log("Add free gift:");
console.log(addFreeGift(cart));

console.log("\n");
console.log("Summarize cart updated:");
console.log(summarizeCartUpdated(cart));
```

Now the bug will surface. When you use the function `addFreeGift()`, you're mutating the `cart` array. It will always have at least *one* `discount` if there are more than *two* items. Even though you're assigning the return value (the `cart` with added gift) to a new variable, you've mutated the original `cart` array. Any time someone has a cart with more than three items and one `discount`, they'll get an *error*.

If this had a test, maybe it would be an easy fix. If there's no test, who knows how long before customer service gets an angry email.

You might notice the problem with a lot of these examples is that the mutation happens in a separate function. Good catch! In fact, that's exactly the reason why mutations can be so dangerous. When you call a function, you should trust that it won't change any supplied values. Functions that have no side effects are called **"pure"** functions, and that's what you should strive to achieve.

It can be even more confusing when you return a value from a function even though you mutated the input. A developer who comes through later will likely assume that the original values haven't changed given that the return value is the one with the update. In this case, they'd be wrong. The input value was also changed.

Fixing the problem using the spread operator

Time to fix the problem. It's so incredibly simple—you should immediately understand why the *spread operator* became so popular.

```
const cart = [
  {
    name: 'The Foundation Triology',
    price: 19.99,
    discount: false,
  },
  {
    name: 'Godel, Escher, Bach',
    price: 15.99,
    discount: false,
  },
  {
    name: 'Red Mars',
    price: 5.99,
    discount: true,
  },
];

const reward = {
  name: 'Guide to Science Fiction',
  discount: true,
  price: 0,
};



function addGift(cart) {
  if (cart.length > 2) {
    return [...cart, reward];
  }
  return cart;
}

function summarizeCartSpread(cart) {
  const cartWithReward = addGift(cart);
  const discountable = cart.filter(item => item.discount);
  if (discountable.length > 1) {
    return {
      error: 'Can only have one discount',
    };
  }
  return {
    discounts: discountable.length,
    items: cartWithReward.length,
    cart: cartWithReward,
  };
}

console.log("Add gift:");
console.log(addGift(cart));
console.log("\n");
console.log("Summarize cart using spread:");
console.log(summarizeCartSpread(cart));
```

All you need to do is take the current array and spread it into square brackets, tacking the newest item on at the end.

In essence, all you're doing is rewriting the contents as a list. Note that this is a brand new array so there's no way we could possibly change the original array. We're just reusing the contents to make a new array.

```
const titles = ['Moby Dick', 'White Teeth'];
const moreTitles = [...titles, 'The Conscious Mind'];
console.log(moreTitles);
```

What I love most about creating new arrays this way (and I'm sure you will, too) is that you can forget so many methods. You won't need them anymore.

> Quick! How do you add a new item to the start of an array? How do you make a copy of an array?

Hide Hint

It's different than assigning the same array to a new variable.

Did you have to look them up? Don't worry, I still do, too. I mean, who could remember that `slice()` is a function to make a copy. Here they are with the *spread* replacement:

```
// Add to beginning.
const titles = ['Moby Dick', 'White Teeth'];
titles.unshift('The Conscious Mind');
console.log("Titles after using unshift: " + titles)
const moreTitles = ['Moby Dick', 'White Teeth'];
const evenMoreTitles = ['The Conscious Mind', ...moreTitles];
console.log("Adding titles using spread operator: " + evenMoreTitles);

// Copy
const toCopy = ['Moby Dick', 'White Teeth'];
const copied = toCopy.slice();
console.log("Copying titles using slice: " + copied)

const moreCopies = ['Moby Dick', 'White Teeth'];
const moreCopied = [...moreCopies];
console.log("Copying titles using spread operator: " + moreCopied);
```

And of course, to repeat a point from earlier, you're signaling your intention to return an array. Another developer may not remember that `slice()` creates a new array, but when they see the square brackets, they'll know exactly what they'll get.

In the next tip, you'll see how creating copies of arrays can prevent problems when you must use methods that mutate arrays, such as `sort()`.