Tip 18: Check Data Quickly with the Ternary Operator

In this tip, you'll learn how to avoid reassignment with the ternary operator.

We'll cover the following

The ternary operator

By now, you may have noticed that I love simple code. I'll always try to get an expression reduced down to the fewest characters I can. I blame a former coworker who reviewed some code I wrote at one of my first jobs.

```
const active = true;
if (active) {
    var display = 'bold'
} else {
    var display = 'normal'
}
console.log(display)
```

He took one glance and casually said, "You should just make that a ternary."

"Of course," I agreed, not quite sure what he was talking about. After looking it up, I simplified the code to a one-line expression and my code has never been the same.

```
const active = true;
var display = active ? 'bold' : 'normal';
console.log(display);
```

Why use the ternary operator?

Chances are you've worked with *ternary operators* before. They're common in most languages, and they allow you to do a quick *if/then* check. (Although they aren't exclusively for this purpose, it is by far the most common usage.)

If the ternary operator isn't new, why should it interest you? In returning to some of the larger themes we've been exploring, ternary expressions allow your code to be not just *more simple* as I've mentioned, but also more *predictable*. They do this because they cut down on the number of variables that are being reassigned.

Besides, with new variable types, we hit some problems with excessive `if`/`else` statements. If you try to check a variable and you're using a block-scoped variable, you won't be able to access the variable outside of the check.

Example: Without the ternary operator

```
const title = 'manager';

if (title === 'manager') {
    const permissions = ['time', 'pay'];
} else {
    const permissions = ['time'];
}
console.log(permissions);
```

Now you're forced to either use `var`, which is accessible outside the block scope, or you have to define the variable with `let` and then reassign it inside the `if`/`else` block. Here's how it would look with the assignment before the block:

```
const title = 'manager';
let permissions;
if (title === 'manager') {
    permissions = ['time', 'pay'];
} else {
    permissions = ['time'];
}
console.log(permissions);
```

Before `let` and `const`, you didn't have to worry so much about when variables were created. Now, in addition to excessive code, there's a potential for scope conflicts.

Ternary expressions solve these problems. Clearly, they cut down on a lot of extra code. But they also allow you to be more predictable by assigning a value directly to `const`. How could you rewrite the preceding code to use const and a ternary?

Example: With the ternary operator

```
const title = 'manager';
const permissions = title === 'manager' ? ['time', 'pay'] : ['time'];
console.log(permissions);
```

Much cleaner and you now have a predictable value.

There's one caution you should keep in mind: Though you can chain multiple ternary expressions together, you should avoid doing so. Imagine that there's another user type called supervisor that couldn't see the pay rate but could authorize overtime. You might be tempted to just add another ternary expression. What's the harm, right?

```
function getTimePermissions(title) {
    const permissions = title === 'supervisor' || title === 'manager' ?
        title === 'manager' ?
            ['time', 'overtimeAuthorization', 'pay'] : ['time', 'overtimeAuthorization']
        : ['time'];
    return permissions;
}
console.log(getTimePermissions('manager'));
console.log(getTimePermissions('supervisor'));
```

At that point, the ternary becomes unreadable and loses the value of simplicity. Instead, you should move the check completely out of the block into a standalone function (with a nice test, of course). That way, you can still use const without worrying about excessive code.

```
function getTimePermissions({ title }) {
    if (title === 'manager') {
        return ['time', 'overtimeAuthorization', 'pay'];
    }
    if (title === 'supervisor') {
        return ['time', 'overtimeAuthorization'];
    }
    return ['time'];
}
const permissions = getTimePermissions({ title: 'employee' });
console.log(permissions);
```

There's no harm in making short functions that have a single non-abstract purpose. In fact, it's a good step to writing clean code. You still get the value of assigning the return value to const and everything is clear and readable. Ternary expressions can simplify things, but use them when they add value and go back to standard if blocks if they create too much ambiguity.

Q

What will be the output of the following code?

```
let price = 180;
let answer = (price < 50) ? "Cheap" :
             (price < 100) ? "Economical" :
             (price < 150) ? "Expensive" : "Overpriced" ;
console.log(answer);
```

Retake Quiz

In the next tip, you’ll make quick data checks even easier with short-circuiting.