

Tip 3: Isolate Information with Block Scoped Variables

In this tip, you'll learn how `let` prevents scope conflict in `for` loops and other iterations.

We'll cover the following ^

- Block scoped variables
- Lexically scoped variables
 - Example
 - Using `var`: Example
- Using `let`: Example

At one point or another, every developer will make the mistake of capturing the wrong variable during a `for` loop. The traditional solution involves some pretty advanced JavaScript concepts. Fortunately, the `let` variable declaration makes this complex issue disappear.

Block scoped variables

Remember, when you use a *block-scoped* variable declaration, you're creating a variable *that's only accessible in the block*. A variable declared in an `if` block *isn't* available outside the curly braces. A variable declared inside a `for` loop *isn't* available outside the curly braces of the `for` loop. But that doesn't mean you can't access variables declared outside a *function*. If you declare a block scope variable at the top of a function, it is accessible inside the block.



Lexically scoped variables

If you declare a *lexically scoped* variable, however, it's accessible *anywhere* inside a function. A variable created inside an `if` block can be accessed anywhere else in the function. In fact, you can even access a variable before it was declared because of a compile process called **hoisting**.

Example

If that all seems too abstract, that's fine. It's easier to understand in practice.

Chances are, if you've encountered a lexical scope issue before, it probably occurred when you were adding a `click` function to a series of **DOM** elements:





Output
HTML
<ul style="list-style-type: none">• Say Zero• Say One• Say Two
<div></div>

See the output code of this code and try clicking on one of the list elements. You'll find that every click will give the same result: `3`.

It's tempting to think this is a browser bug, but it's actually more related to how JavaScript assigns variables. It can happen anywhere, even in regular JavaScript code. Let's look at how this issue can occur even in plain JavaScript without DOM manipulation.

If you paste this code into a browser console or a REPL, you'll see the same problem.

```
function addClick(items) {
  for (var i = 0; i < items.length; i++) {
    items[i].onClick = function () { return i; };
  }
  return items;
}
const example = [{}, {}];
const clickSet = addClick(example);
console.log(clickSet[0].onClick());
```



 Using REPLs

No matter which array element you try, you'll get the same result.

Why is this happening?

The problem again is *scope*. Variables assigned with `var` are *functionally* scoped (which, again, is technically referred to as *lexically* scoped). That means that they'll always refer to the **last** value they're assigned within a function.

Using `var`: Example

When you set a new function on **line 3** in the preceding example, you're saying to return the value `i` whatever it may be at the time you call the code. You are not saying: return the value of `i` at the time it's set. As a result, because `i` belongs to the function, the value changes on each loop iteration.

The traditional solution is complicated, and it can confuse even the most experienced JavaScript developers.

```
function addClick(items) {
  for (var i = 0; i < items.length; i++) {
    items[i].onClick = (function (i) {
      return function () {
        return i;
      };
    })(i);
  }
  return items;
}

const example = [{}, {}];
const clickSet = addClick(example);
console.log(clickSet[0].onClick());
```

It involves a combination of **closures** (creating a variable inside a function for another function to use), **higher-order** functions (functions that return other functions), and **self-invoking** functions. If you don't understand that, it's fine. You'll learn more about higher-order functions in [Tip 34](#), Focused Parameters with Partially Applied Functions.

Using `let`: Example

Fortunately, you don't need to understand these higher concepts quite yet. If you rewrite the preceding code using `let`, you'll get the same results without the extra code clutter. You can test out the following code in a browser console or REPL and you'll get the results you see below:

```
function addClick(items) {  
  for (let i = 0; i < items.length; i++) {  
    items[i].onClick = function () { return i; };  
  }  
  return items;  
}  
const example = [{}, {}];  
const clickSet = addClick(example);  
console.log(clickSet[0].onClick());
```



Looking at **line 3**, you'll notice the only thing you changed is using `let` instead of `var`. Because `let` is blocked scoped, any variable declared inside the `for` block belongs only to that block. So even if the value changes in another iteration, the value won't change on the previously declared function.

In simpler terms, *`let` locks the value during each iteration of the `for` loop.*

Because `let` can do nearly everything `var` can do, it's always best to use `let` whenever you might otherwise use `var`.

I hope this gave you some ideas for how to declare variables. You will find in upcoming tips that variable declaration is so important that you may want to restructure whole code blocks to keep declarations clear and predictable.

In the next tip, you'll look at how to transform data to readable strings using template literals.