Immediately Invoked Function Expressions

In this section, we'll cover a concept commonly asked about in JavaScript interviews. We'll see how immediately invoked function expressions allow us to overcome the problems of scope pollution and unwanted variable behavior.

Introduction

As mentioned in the previous lesson, one of the goals of functional programming is to reign in the scope of variables. We want variables to only be accessible in their small, local scope, and only where they're needed. This reduces the chance of bugs, since we can quickly find all pieces of code that a variable is connected to.

This section discusses one way to implement this idea. We're learning a tool to reign in variable scope and keep variables as localized as possible.

Scope Pollution

Writing code often pollutes the global scope. Depending on what we're doing, we can have dozens of variables in our file and it's easy to lose track of them.

Eventually, when our code gets complex, we'll start running into errors where we've named variables the same thing. They're incredibly hard to debug and annoying to deal with, as we have to be even more creative in coming up with variable names.

IIFEs

There's a way around this and it's called an immediately invoked function expression. The basic format looks like this.

```
(function(){/* some code here */})()
```

What we're doing here is wrapping all of our code inside a function. This keeps all of our variables and functions in the scope of that function.

Attempting to access variables outside won't work and our global scope is empty. We've essentially "hacked" ourselves a new local scope.

This new function is evaluated immediately, as indicated by the `()` at the end of the function.

Since our function doesn't have a name, there is literally nothing on the global scope. No matter what we write in the function, our global scope is clean.

There are a few other ways to write an IIFE that you might see.

```
(function(){})();
(function(){}());
!function(){}();
~function(){}();
+function(){}();
-function(){}();
```

All of these things do something to the function: they turn it into an expression to be evaluated. If we were to write

```
function(){}()
```

we'd get a syntax error. We can't invoke a function declaration directly.

This discussion leads us to an interesting interview question. Now that you know what an IIFE is, see if you can apply it to this problem.

The IIFE Interview Question

Required Reading

If you're not familiar with `setTimeout`, it might be a good idea to skip ahead and go to the Asynchrony section. You'll find some lessons that can introduce you to asynchronous code. It's mostly a stand-alone section so skipping ahead shouldn't make the content too difficult.

The Question

A common interview question is the following.

What will this code print out?

```
for (var i = 0; i < 5; i++) { // We are explicitly using `var` for a reason
    const time = i * 1000;
    setTimeout(function() {console.log(i);}, time);
} // -> ?
```

`setTimeout` doesn't work correctly in educative.io's code editor, but what happens here is that the number `5` is logged after 0, 1, 2, 3, and 4 seconds. Try it out in another JavaScript evaluator.

This happens because by the time `i` is printed, the loop has finished and its value is `5`. `i` is present in the global scope and not limited to the loop. It's this global `i` that the `setTimeout` call is using. Since we make it wait, `i` has already become `5` by the time they each run.

The next part of this question is:

How can we fix it, so that it logs 0, 1, 2, 3, 4?

The answer is an IIFE. By creating a closure inside an IIFE, we can permanently set `i` inside the function to the value we need.

```
for (var i = 0; i < 5; i++) {
    (function(num) {
        const time = num * 1000;
        setTimeout(function() {console.log(num);}, time);
    })(i);
}
// -> 0
// -> 1
// -> 2
// -> 3
// -> 4
```

If this is flying over your head, that's perfectly fine. Come back to it. This is very challenging. It tests variable hoisting, closures, IIFEs, and asynchronous code. But you have the skills to understand the solution.

We're immediately invoking the outer anonymous function, passing in `i`

We're immediately invoking the outer anonymous function, passing in `i`. Inside that function, `num` is set to that value of `i`.

Since `num` is locally scoped inside the IIFE, it keeps its original value even when `i` changes. Every run through the loop calls the IIFE again and creates a new instance with a brand new scope and locked-in value of `num`.

`setTimeout` now works as expected, using the locally-scoped `num`.

That's it.

In the next lesson, we'll dive into this example further and show different ways of solving this problem using functional programming.